Modeling and Simulation of Cooperative Multi-Agents in Transactional Database Environments

Khaled Nagi

Institute for Program Structures and Data Organization, Universität Karlsruhe.
Am Fasanengarten 5,
D-76131 Karlsruhe, Germany.
+49-721-608-7336

nagi@ira.uka.de

ABSTRACT

Multi-Agent Systems (MAS) are finding their way into large-scale information systems. Unfortunately, current prototypes often lack the stability and robustness necessary for real-world deployment. We propose transaction mechanisms as a possible remedy. For this, we divide MAS into two layers:(1) a planning layer, whose actions are idempotent, do not cause any real-world side-effects and are therefore easily recoverable, and (2) an execution layer, which is responsible for all real-world actions and runs under transactional protection to achieve robustness.

The execution layer consists of special Execution Agents, one for each agent in the planning layer. An Execution Agent is responsible for the robust execution of the plan developed by its peer planning agent. For this purpose, it implements an extended database transaction model - which models the execution dependencies between agent actions - and provides recovery mechanisms to deal with possible according to a predefined contingency behavior within the agent plan.

To assess the overhead induced by the plan execution layer, we built a simulator to analyze the scalability and the performance of the system under various workloads, resource supplies, and disturbance frequencies. In contrast to our previous simulation work, we concentrate in this paper on modeling cooperative - instead of antagonist - agents and on long-running plan executions. After a brief outline of the transaction model, we describe the simulation model and report on the experimental results of introducing two types of disturbances into a system of cooperating agents.

Keywords

Modeling and simulation, performance analysis, planning and execution, robustness, transaction management.

1. INTRODUCTION

Multi-Agent Systems (MAS) have been traditionally employed in Artificial Intelligence to solve highly complex distributed planning problems (see [12] for a good introduction). Today, this technology is increasingly applied to information systems. Their ever-growing complexity and the difficulty to foresee all potentially arising disturbances makes them an inviting test bed for MAS. As can be seen in [6], many research efforts are dedicated to deploying this emerging technology in the fields of intelligent information retrieval, web assistants, information trading, matchmaking, etc. Yet, this technology is slow to find its way into large-scale information-rich applications, in which actions are to be automatically *executed* instead of just providing decision support services. The reason is the lack of robustness in existing MAS. By robustness, we mean that both individual agents and the MAS as a whole overcome disturbances, failure, or uncontrolled interactions by reaching well-defined states.

In order to combine robust execution of agent actions with the planning power of agents, we need to represent the plan together with its possible contingency behavior in one *structure*. In [9], we argued for the use of open-nested transaction trees to build this structure. In our solution, we entrust the execution of each of these trees to a special component, called the *Execution Agent*, which is also responsible for the robustness of execution.

1.1 System Architecture

Embedding the *Execution Agent* in a general MAS architecture is illustrated in Figure 1. The MAS is situated in a world (environment) that is represented by a federation of databases. Agents can perceive their environment by reading the databases and can modify it by writing to the databases. The agents are divided into groups and the members of each group are cooperating to achieve a shared common goal. Each agent is actually divided into two physical entities: a *Planning Agent* and an *Execution Agent*. The *Planning Agent* develops the agent-specific part of an overall shared plan based on the common goal. This requires, of course, cooperation between the different planning agents. Then, each Planning Agent delegates its part of the shared plan to a peer *Execution Agent*. Each local plan of this shared plan defines a set of simple agent actions and a set of execution dependencies between them. The plan is formulated as a *transaction tree*, as described in [9]. In these trees, each simple agent action is encapsulated in one ACID transaction, i.e., transaction satisfying the *Atomicity*, *Consistency*, *Isolation*, and *Durability* conditions [2], that access the database. The cooperation between the individual agent plans appears in a series of coordination primitives connecting the transaction trees at the execution layer. Through the definition of various control parameters and control flow rules, the contingency is also defined within the transaction tree, thus capturing both normal and contingency behavior in one execution structure.

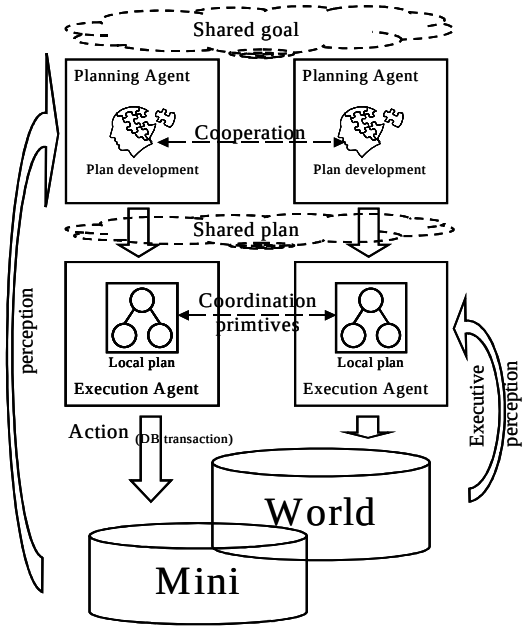

Figure 1. A MAS architecture with Execution Agents

The Execution Agent, illustrated in Figure 2, receives a transaction tree, either complete or incrementally from its peer Planning Agent. The *Agent Transaction Manager* (ATM) module is responsible for the correct and robust execution of these actions. The simple agent actions are submitted to the underlying databases through the *Database Interface* module in the form of ACID transactions. This module is responsible for establishing the connection to the database, submitting the SQL commands, receiving the execution results of the transaction, and coupling robustness mechanisms of the Database Management System with those of the Execution Agent. On the other hand, the *Communication Interface* module is responsible for communication with other Execution Agents of the same agent group. Based on the incoming messages, it reports the status of the relevant parts of the various transaction trees to the ATM.

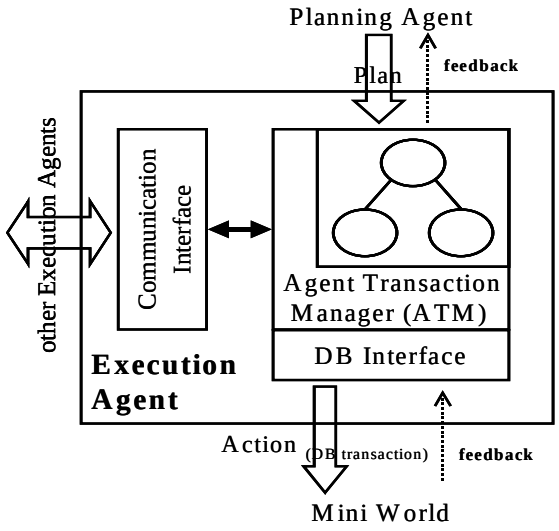

Figure 2. The Execution Agent

In order to keep the Execution Agent open for integration with different MAS, we designed the interface of the Execution Agents to be compliant with the FIPA Agent Communication Language standards [5]. This allows diverse planning agents to use the Execution Agents. However, a key to the success of this architecture in large-scale information systems is its ability to scale.

To quantitatively assess the scalability of the system and to have a better understanding of the behavior of the transaction trees and their interactions, we built an event-driven simulator around it. This is very important in evaluating the overall quality of solutions provided by the MAS. For this reason, we present the simulator as a tool and encourage MAS developers to use it to evaluate their systems before the actual deployment. In our previous work presented in [8], we restricted our analysis to antagonist agents competing on the same database. In this paper, we concentrate on cooperative agents executing a shared plan. More specifically, we investigate the effect of executing contingencies in coordinated transaction trees in case of disturbance on the overall performance of MAS. Contingencies are executed due to failures during the simultaneous execution of multiple Execution Agents working on the same shared plan *or* due to changes in the mini world during long-running plan execution. Such execution usually spans the time interval between the point of decision making till the point of actual execution. For concrete examples refer to Section 4.

The remainder of the paper is organized as follows. Section 2 contains an overview of the agent transaction model. In Section 3, we describe the main contribution of this paper, which is the simulation model and the simulator. Section 4 analyses the simulation results. Finally, Section 5 summarizes the paper.

2. THE AGENT TRANSACTION MODEL

We use a model based on open nested transaction trees introduced in the early 80s for database management systems [11]. A hierarchical transaction model seems to be best suited for representing planning strategies [10], since the majority of planning algorithms is based on hierarchical decomposition (for a good overview, please refer to [1] and [12]). Under this model, a transaction can launch any number of subtransactions, which, in turn, can launch any number of subtransactions, thus forming a transaction tree. In general, a transaction cannot commit unless all its children are committed. However, if one of the children fails, its parent does not have to abort. It has the choice between: *ignoring* the failure (a non-vital transaction), *retrying* the subtransaction, or *aborting*. Subtransactions may commit or abort independently and appear atomic to other subtransactions. A committed subtransaction makes its results available to other subtransactions in other transaction trees as soon as it commits. The global correctness criterion maintained by the Execution Agent is: *either* the complete execution of the transaction tree *or* it appears to not have executed at all. Thus, in case of transaction abort, a *backward recovery* must take place, in which committed subtransactions of the aborted subtree are compensated by executing so-called *compensating transactions*, whose role is to undo the effects of the original subtransactions.

Figure 3 is an example of a transaction tree. A *Control Node* is a non-leaf node. It is executed by the ATM of the Execution Agent according to the control parameters supplied to it by its peer Planning Agent. Typical control parameters include the *number of retries* in case of failure and the *time interval* between retries. For a detailed description of the parameters, please refer to [7]. Dur-

ing normal execution, the children of a control node can execute either in *parallel* or *sequentially*. An *Action Node* is always a leaf node. It represents a simple agent action, which is submitted through the DB interface module to the corresponding database management systems in the form of an ACID transaction. Each of these actions returns either *success* or *fail* to its parent. For each action node, a compensating action is defined. Also, a reassessment action can be defined as will be seen later in this section. The semantic of compensation and reassessment actions, their place within the trees and the triggering rules define the contingency behavior in case of disturbance.

In order to support agent cooperation in implementing the shared plan, the corresponding transaction trees must be coupled in the execution layer. For this reason, we introduce so-called *Synchronization Nodes*. They represent the *basic coordination primitive* between transaction trees and their operation is defined through *Event*, *Condition*, *Action* (ECA) rules. There are two types of Synchronization nodes: a *sender* node and a *receiver* node, situated in different transaction trees. The sender node sends a message to the receiver node if an *event* occurs, which is the change in the execution state of the sender node and a certain *condition* is met. A typical condition would be the commitment of a transaction subtree. The receiver node waits for the arrival of the message and executes the *action* defined in the message. The action is a command to change the state, usually a commit or an abort, of the receiver node. The presence of the synchronization nodes violates the strict hierarchical structure of the transaction trees, since receiver nodes are now controlled by *two* nodes: *directly* by their parents in the tree and *indirectly* through the message received from their sender nodes. A typical coordination problem arises when the synchronization node receives a commit message from its sender node and, as a result, commits. At a later point, the foreign subtree, corresponding to the sender node is to be compensated due to a failure. This logically invalidates the status of the receiver node and all subsequent nodes in the tree. In this case, the ATM starts a *forward recovery* process. It tries to *reassess* all subtransactions depending on (i.e., occurring after) the receiver node. By reassessment, we mean launching a reassess transaction that primarily undoes the effect of the original transaction then retrying it in case that the guarding condition for this transaction is violated. A guarding condition specifies the limit for the change in the mini world that would still *not* invalidate the effects of a committed transaction. Reassessment transactions are similar in concept to compensation transactions. If this forward recovery fails, a backward recovery takes place.

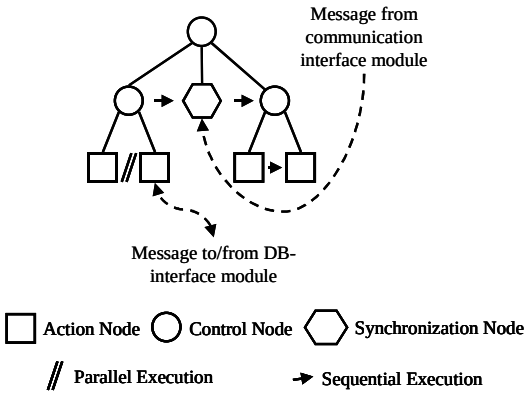

Figure 3. A transaction tree

Figure 4 illustrates a simplified state transition diagram that applies to all node types. At the beginning, the transaction node is in the *Waiting* state until the ATM starts it. It then moves to the *Executing* state. According to the result of the execution (e.g., success or fail of an action node or the final state of the children of a control node), the node moves either to the *Committing* state or to the *Aborting* state. If aborted, the node waits for a time interval before moving again to the Executing state as long as the number of retries is not yet exhausted. A committing node moves to the *Compensating* state if the *must undo* flag is set in its control parameters and its parent node fails. After compensation, it waits again for execution by the ATM. Similarly, if a forward recovery takes place and the committed transaction is affected, it moves to the *Reassessing* state, then to the Executing state for a new retry. Otherwise, the committed node remains in its current state until all root nodes in the cooperating agent group commit. Here, a procedure similar to the 2-Phase Commit protocol [2] is instantiated to coordinate their terminations. Then, all the committed nodes in the tree enter the *Terminating* state. However, if the root node fails and exhausts all its retries, all of its descendents enter the *Abandoning* state.

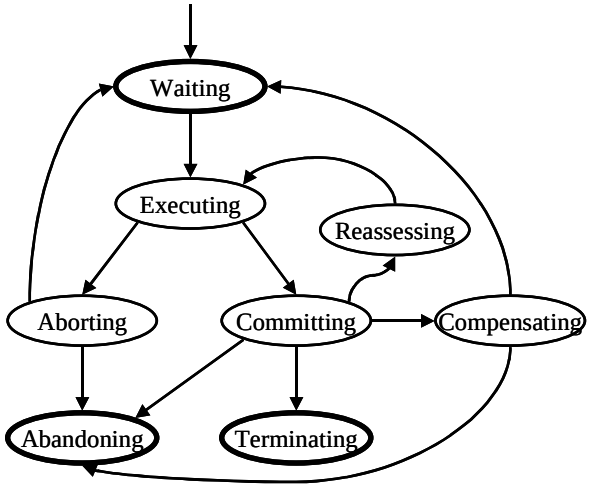

Figure 4. State transaction diagram for a transaction node

3. THE SIMULATION MODEL

In [8], we analyzed the behavior of a system consisting of a growing number of antagonist agents, each realizing its own goal while operating on the same database. In this paper, we model and simulate cooperative multi-agents. More formally, according to [4], we handle the case of coordinated collaboration between agents with collective conflicts over resources. We concentrate on the scalability and the analysis of system behavior in face of executing contingencies occurring either due to failures in the execution of coordinated transaction trees or due to changes in the mini world that invalidate the decision taken by some Execution Agent during a long-running plan execution. As illustrated in Figure 5, we replace the levels above and underneath the execution layer with two simulated models and a disturbance simulator. In the following subsections, we describe these models and the transaction tree model executed by the Execution Agents.

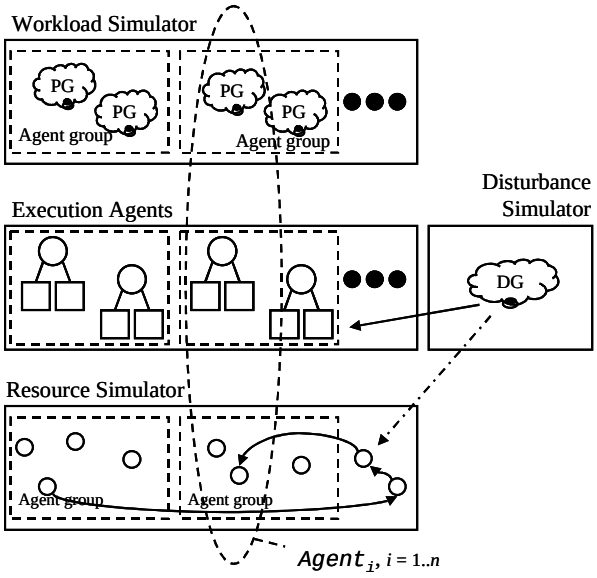

Figure 5. The simulation environment

3.1 System Parameters

3.1.1 The workload model

The agent population is divided into agent groups. Each group contains a fixed number of agents that work on a shared plan. Each Planning Agent in the group is represented by a *Plan Generator* (PG) module. This module generates a transaction tree and submits it to its peer Execution Agent. When the Execution Agent finishes executing the transaction tree, it waits until all other Execution Agents in the group also finish before requesting a new transaction tree from the PG. In those simulations, where plan reassessments due to changes in the mini world are considered, committed trees are transferred to a list of pending trees as illustrated in Figure 6, until the lifetime of the plan execution comes to end. Meanwhile, the Execution Agent can process other transaction trees as long as no reassessment of a pending tree is needed.

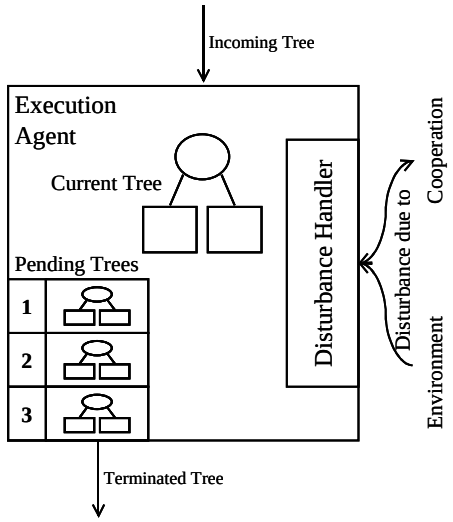

Figure 6. Simulation Model for an Execution Agent

Three parameters specify the tree configuration as illustrated in Figure 7.

Average Number of Children (C): is a uniformly distributed random variable between 1 and 2*C*; determining the number of children of a control node.

Probability of Simple Transactions (S): is a Bernoulli trial with probability *S* that a child is an action node.

Probability of Parallel Execution (P): is a Bernoulli trial with probability *P* that the children of a control node are executed in parallel.

We notice the absence of explicit synchronization nodes in the generated transaction trees. During normal operation, they act only as means for synchronization and show an effect similar to normal action nodes. In case of disturbances, the important role of their ECA rules is already modeled through the disturbance simulator as will be shown in Subsection 3.1.4.

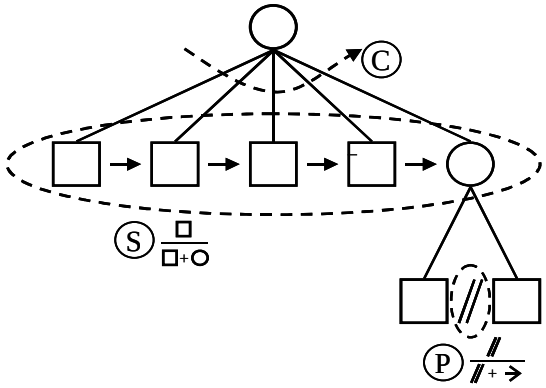

Figure 7. Parameters for the transaction tree

3.1.2 The transaction tree model

As in [8], we adopt a persevering strategy to yield more concrete and quantitative results for our chosen performance indices. This means the following settings of control parameters of transaction tree nodes. All nodes must be undone in case of backward recovery and all dependent action nodes must be reassessed in case of forward recovery (i.e., the guarding conditions always fail during the reassessment phase). In the experiments summarized in Section 4.1, the number of retries of action nodes is set to 2 in normal, compensation, and reassessment modes. Control nodes are retried 5 times. This setting helps analyzing the effect of compensation of subtrees on the other Execution Agents of the same group; and hence the effect of propagating contingencies in coordinated transaction trees. In the experiments summarized in Section 4.2, the number or retries of action nodes is raised to 5 in order to avoid unnecessary aborts of control nodes. In both types of experiments, we fix the time interval between successive retries to 60 seconds for all node types in all operation modes.

3.1.3 The resource model

Execution time of database transactions, normally consisting of read and write operations, is represented by an exponential distribution with mean 15 seconds. To determine whether a transaction is to be *committed* or *aborted*, we construct a transaction *serialization graph*. Each executing transaction is represented by a node in the graph. A directed edge indicates a conflict between two

transactions. A conflict occurs if two transactions access the same data object and at least one operation is a write. A transaction is aborted if its introduction results in a cycle in the graph [2]. Edges are added with *three* probabilities: P_1 for nodes belonging to the same agents, a higher P_2 for nodes belonging to the same agent group, and a highest P_3 for nodes belonging to different agent groups. This reflects an assumption about the planning algorithm: Planning Agents tend not to submit conflicting actions simultaneously. This applies to actions specified in plans belonging to the same agent group and to a greatest extent to actions belonging the same agent. P_1, P_2 and P_3 are set to 0.03, 0.06 and 0.09 respectively. Sources for these values as well as mean execution time are taken from the literature in database modeling such as [3], where both the database and transaction sizes are reduced to a small size, while preserving the relationship between them. Figure 7 illustrates a serialization graph. Each node has an *agent group identifier* (`AgGpID`), an *agent identifier* (`AgID`), and a *transaction identifier* (`TID`).

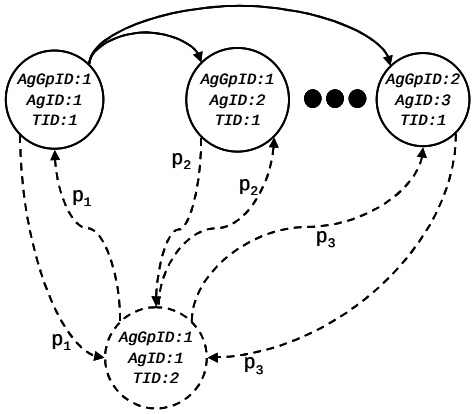

Figure 8. Inserting a node into the serialization graph

3.1.4 The disturbance model

While the resource model reflects failures due to conflicts in the underlying database, the disturbance model describes failures resulting from two categories of dependencies. The first one is caused by the synchronization nodes and results from coordinated transaction trees of Execution Agents in the same agent group; the second category results from changes in the mini world during long-running plan executions. A typical pattern of such execution is a phase of intensive actions, in which the necessary decisions are taken and registered in the database, followed by a quiet phase in which changes in the mini world may not invalidate these decisions [10]. At the end, another active phase takes place, in which these decisions are implemented in the mini world.

In order to model the first category, an Execution Agent making a backward recovery of a subtree reports the number of affected simple actions to the disturbance simulator. The disturbance simulator, in turn, *simulates* ECA rules for synchronization nodes typically existing in coordinated transaction trees. It considers all committed simple actions at Execution Agents belonging to the same agent group for reassessment. It introduces reassessment for each committed simple action according to a Bernoulli trial with a probability directly proportional to the ratio between the number of affected simple actions of the subtree and the total number of committed simple actions at all members of the agent group; the proportion constant being F_C.

As for the second category, we built a Disturbance Generator (*DG*) module. In time intervals following an exponential distribution with mean T_1, it simulates disturbances occurring due to changes in the mini world. In our runs, we fix T_1 to 300 seconds, which has the same order of magnitude as the response time of a transaction tree. These disturbances launch the contingency behavior in the transaction trees and cause the reassessment of committed simple actions that are either part of the current transaction tree or part of the list of pending trees. This is done according to another Bernoulli trial with probability P_D. In this modus, a committed transaction tree has to remain pending for a certain time interval before being terminated. This time interval represents the total lifetime of the plan and follows also an exponential distribution with mean T_2.

3.2 Performance Indices

We consider the same performance indices as in [8]. In this section, we include a brief description of these indices for completeness. The first index is the *throughput* accounting for the number of terminated actions per agent per second.

$$Throughput = \frac{\sum \text{terminated simple actions}}{\text{Simulation time} \times \sum \text{agents in the simulation}}$$

The second index is the *response time* of the whole transaction tree. In this index, we also account for the time wasted in aborted, compensated, reassessed, and abandoned actions. We also include time spent by the tree waiting in the pending list of the Execution Agent in our calculation. Formally, we define the response time to be:

$$Response\ Time = \frac{\sum_{i=1}^{\text{number of transaction trees generated in the simulation}} (\text{End time of the root node}_i - \text{Start time of the root node}_i)}{\sum \text{transaction trees generated in the simulation}}$$

In many cases, one transaction specification can serve as compensation well as reassessment. Due to the similarity of the effects of compensation and reassessment, we combine them into one index: the *ratio of compensated/reassessed actions to the terminated/abandoned ones.*

$$Compensation/Reassessment\ Ratio = \frac{\sum \text{compensated simple actions} + \sum \text{reassessed simple actions}}{\sum \text{terminated simple actions} + \sum \text{abandoned simple actions}}$$

This is a very important economical measure because of the high cost generally associated with compensation or reassessment. For example, one gets only a percentage of the full price for a returned theater ticket. A minimization of this ratio is certainly desired. The *ratio of abandoned actions to the terminated/abandoned ones* accounts for the incidents, in which the Execution Agent fails to execute a plan and returns control back to its peer Planning Agent. A low value of this metric is a good measure for achieving the design goal of separating planning from the details of execution.

$$Abandon\ Ratio = \frac{\sum \text{abandoned simple actions}}{\sum \text{terminated simple actions} + \sum \text{abadonned simple actions}}$$

The last performance index is the *ratio of aborted actions to the terminated/abandoned ones.* It accounts for work lost due to conflicting actions. Here also, a lower ratio is desired.

$$Abort\ Ratio = \frac{\sum \text{aborted simple actions}}{\sum \text{terminated simple actions} + \sum \text{abadonned simple actions}}$$

4. SIMULATION RESULTS

In the simulation study, we vary the simulation parameters that trigger the contingency behavior either in coordinated transaction trees or in long-running plan executions. We repeated the experiments for different setting of workload and resource supplies in order to explore the domain space. This also helped in carrying out a sensitivity analysis of the system in the face of variation in workload and resource parameters. Details of this sensitivity analysis remain outside the scope of this paper due to space limitations. However, we present two representative sets of results in the following subsections aided by a set of simple figures illustrating the behavior of the performance indices. In both sets, we set the values of *P*, *C*, and *S* to 100%, 5, and 0% respectively. This results in generating flat transaction trees having a root control node and 5.5 simple actions on the average that must be executed sequentially. We also give a small example scenario for each set of experiments.

4.1 Effect of Disturbance on the Execution of Coordinated Transaction Trees

4.1.1 Example

In this subsection, we investigate the effect of disturbances in coordinated execution of transaction trees. Consider the case of a travel assistant agent. User *X* wants to travel to place *A*, and if her old friend *Y* is going to be in Place *B*, she would go and visit her afterwards in *B*. Otherwise, user *X* will prefer to go to place *C*. Intuitively, its Execution Agent would receive a plan that reserves a trip to *A*, insert a synchronization node waiting for a confirmation from the Execution Agent of user *Y*. Upon the receipt of the message, it then executes the subtree reserving a trip from *A* to *B*. If, for any reason, user *Y* backs-off from going to place *B*, the Execution Agent of user *X* should reassess the subtree reserving the trip from *A* to *B*. It would cancel it and reserve for a trip from *A* to *C*. This ought to be done without the intervention of the Planning Agent of user *X*, since the contingency behavior is already defined within the transaction tree and the semantic of its synchronization nodes.

4.1.2 Results

In this set of experiments, we fix the size of the agent population to 120 agents and change the size of the agent groups from 1 to 24; as can be seen along the *x*-axis of Figure 9 through Figure 13. We repeat the experiments for increasing values of F_C.

First, we start with $F_C = 0$, representing a theoretical optimum in which cooperating agents work without any negative side-effects between the members of the agent group. This setting outlines the positive effect of having several agents solving a problem. An improvement in all performance indices is observed. This is easily accounted for if we consider the job mix in the resource simulator. With the increase on the size of the agent group, directed edges of the transaction serialization graph tend to be added more with probability P_2 than with probability P_3. Since P_2 is less than P_3, this leads to less cycles in the serialization graph; and hence less conflicts; less aborts, less compensations and less abandoning of agent actions.

With the increase in F_C, we notice a growing effect, namely, that's of agents backing-off their plans and affecting the execution of other coordinated transaction trees in the same agent group. This propagation of contingencies works against the positive effect of agent cooperation. This leads to the deterioration of all performance indices. Again, this can be intuitively explained. With the presence of such disturbances (represented by $F_C > 0$), the larger the size of the agent group gets, the higher the chance of propagation of such contingencies gets, as illustrated in the previous example. For the given setting of workload and resource parameters, all performance indices tend to achieve asymptotic values for F_C above 6. Precisely for $F_C = 6$, the throughput decreases by a factor of 250 as the group size increases from 1 to 24. The response time increases from 1000 to 5000 seconds. The compensation/reassessment ratio, the abandon ratio, and the abort ratio are also badly affected.

A point of equilibrium for F_C, where both effects seem to equalize, is slightly different for each performance index. It is about 0.5 for both the throughput and the response time, 1 for both the compensation/reassessment and abort ratio, and 2 for the abandon ratio.

4.1.3 Lessons learned

The results of this experiment confirm the famous tradeoff between the gain in having several agents cooperatively working together on the same problem and the loss represented in the increasing overhead of coordinating their actions. The good news is that, even under hard conditions, all performance indices tend to achieve asymptotic values in our system. MAS designers have to evaluate F_C typical for their workload. Using the results of this type of experiments, they should be able to determine the optimal group size that meets the required performance indices. Or, in other words, given a group size, they can predict the MAS performance.

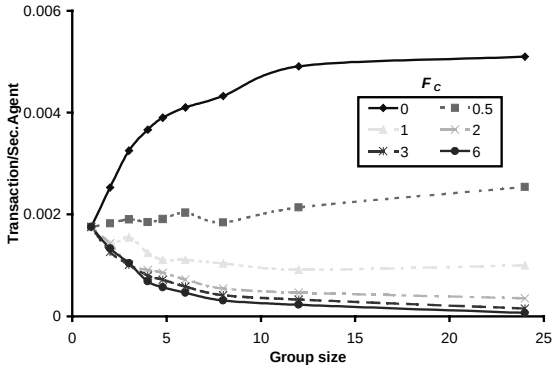

Figure 9. Throughput

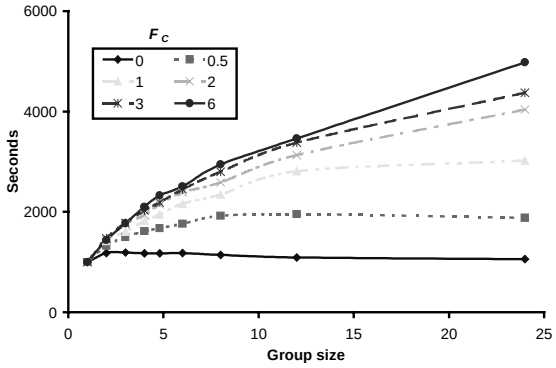

Figure 10. Response Time

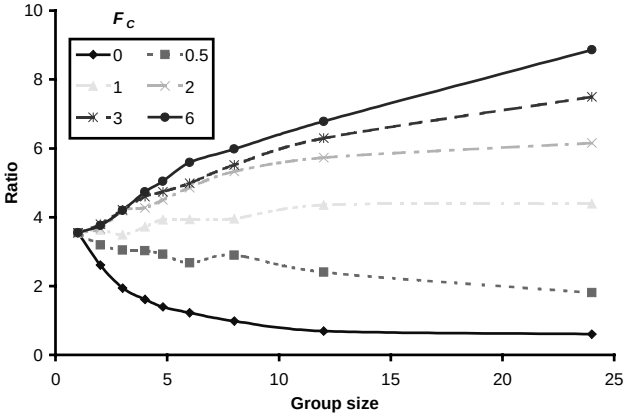

Figure 11. Comp.-Reassess. Ratio

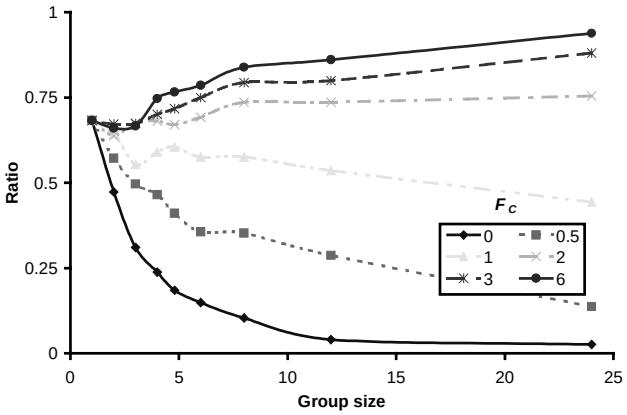

Figure 12. Abandon Ratio

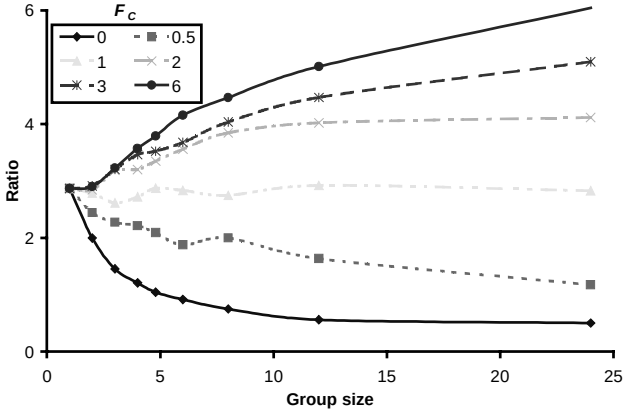

Figure 13. Abort Ratio

4.2 Effect of Disturbance due to Long-Running Execution

4.2.1 Example

In this subsection, we investigate the effects of changes in the mini world during the execution of long-running of agent plans. Consider again the case of a travel assistant booking connection flights from *A* to *B* and from *B* to *C*. Till the actual start of the trip, the plan might be invalidated due to a change in the environment, such as the cancellation of the first flight. The Execution Agent would reassess its transaction tree by booking another flight from *A* to *B* and eventually another one for the trip from *B* to *C* in case the new transit time is too short. Again, this ought to be done without the intervention of its peer Planning Agent since the Execution Agent has already all the needed information for the contingency behavior in the transaction tree.

4.2.2 Results

In this set of experiments, we fix the size of the agent population to 60 agents. Since we are only interested in disturbances resulting from changes in the mini world, no agent groups are formed. Instead, we vary the value of T_2 (the average time interval a committed transaction has to remain pending before termination) from 200 to 2800 seconds, which is about 10 times the value of T_1 (the average time interval between successive disturbances). We repeat the experiments for increasing values of P_D.

First, we start with P_D = 0%, representing the theoretical optimum, in which the disturbances generated by the *DG* module do not effect either the active or the pending transaction trees. As expected, all performance indices remain unchanged, except the response time, illustrated in Figure 15. It increases linearly with the increase in T_2, since the waiting time is also included as part of the overall response time. This experiment serves as a basis of comparison.

Apart from this theoretical optimum, all performance indices deteriorate with the increase in T_2. However, this degree of deterioration strongly depends on the performance index and P_D. Agent throughput, illustrated in Figure 14, degrades heavily once P_D gets above the zero level. With P_D = 50% and T_2 = 200 seconds, the throughput is decreased by a factor of 2.6. This factor becomes 7.7 at T_2 = 2800 seconds. The good news is that the effect of T_2 seems to weaken with the further increase in T_2. This can be observed in the decrease in the absolute value of the gradient of

the plotted curves. This would intuitively lead to a stabilization of the throughput value. On the other hand, the response time, illustrated in Figure 15, seems to be less affected by the increase in P_D. The increase always remains linear and in the increase in the slope of the curve is hardly remarkable.

The deterioration of the other performance indices remains almost linear but seems to be more affected by the increase in P_D than the response time. Both the slope and displacement of the compensation/reassessment ratio, illustrated in Figure 16 are affected by the increase in P_D. Taking P_D = 50% as an example, the slope is more than doubled with the increase of P_D from 5% to 50%. For the point T_2 = 200 seconds and P_D = 50%, the ratio jumps from 0.23 to 1.24. To some extent, the same also applies for the abort ratio illustrated in Figure 18. However, for the abandon ratio, illustrated in Figure 17, only the slope seems to be affected. The displacement remains almost unchanged.

4.2.3 Lessons learned

The results of the experiment confirm the intuitive fact that, in a dynamic environment, decision taken will not always remain valid especially if the time span between the decision taking process and the actual execution is too long. The good news is that for reasonable values of P_D, the system behavior is still acceptable. Moreover, as P_D grows beyond 50%, the system shows an asymptotic behavior, which speaks for the stability of the system.

Since the value of P_D and T_1 are relatively easy to estimate even before the actual deployment of MAS, system designers are encouraged to run such experiments under the expected workload in order to estimate how near to real-time, their plans should be to meet their performance requirements.

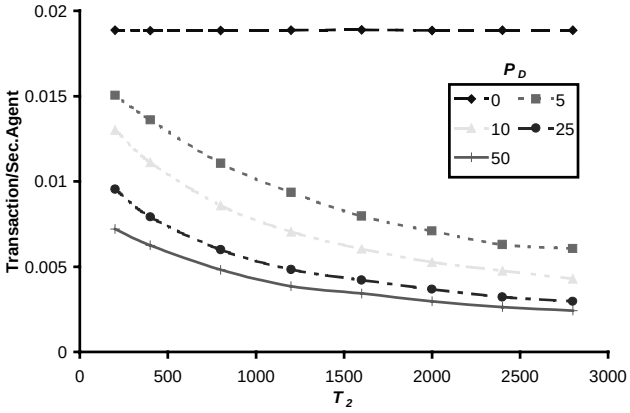

Figure 14. Throughput

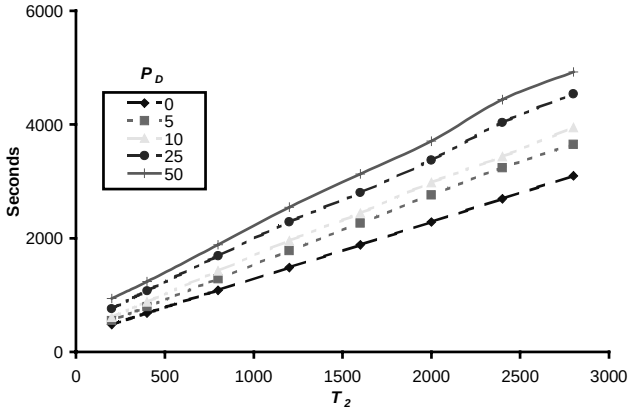

Figure 15. Response time

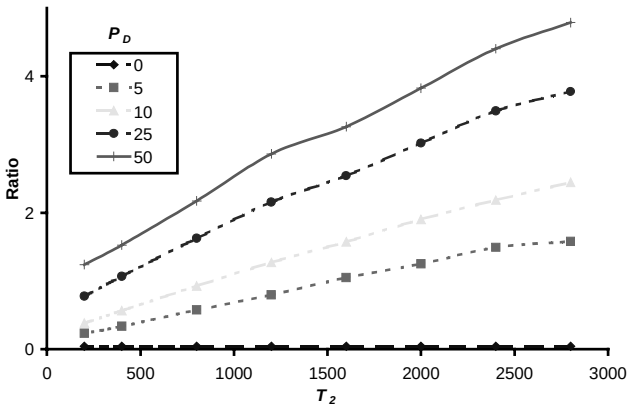

Figure 16. Comp.-Reassess. Ratio

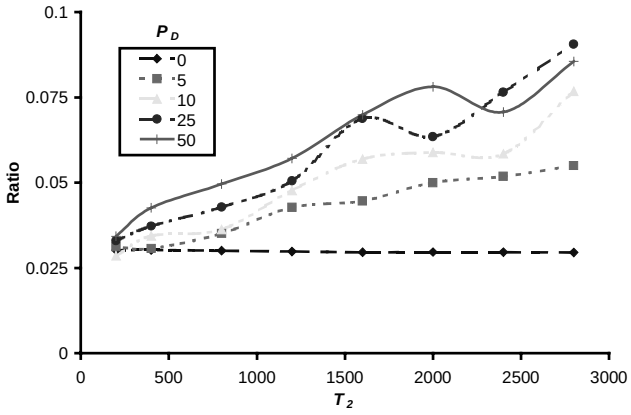

Figure 17. Abandon Ratio

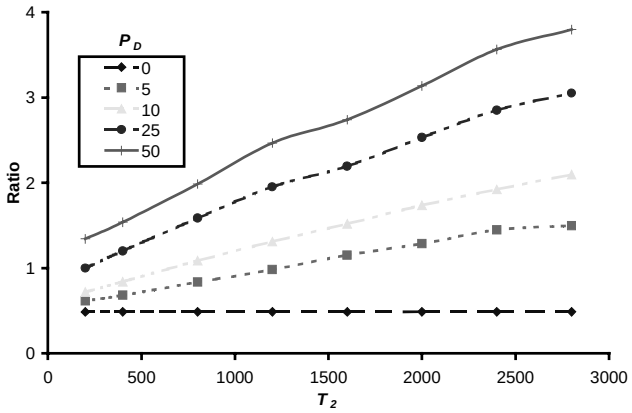

Figure 18. Abort Ratio

5. SUMMARY

In order to increases the robustness of MAS, we entrust the execution of agent actions to special Execution Agents. These agents implement an extended transaction model that formally guarantees robustness of execution. To assess the overhead induced by the Execution Agents, we developed a simulation model and implemented a simulator around them.

In previous work, we restricted our analysis to the case of antagonist agents with self-centered goals. In this paper, we generalize our analysis to include cooperating agents and agents executing long-running plans. For this purpose, we model, simulate, and analyze two categories of disturbances. The first category results in the propagation of contingency among coordinated agent plans. The second category occurs due to changes in the mini world during long-running plan execution. In both categories, a deterioration of performance was observed in comparison to an ideal world without disturbances. However, the degree of deterioration depends on the degree and frequency of the disturbances. With extremely high disturbance, the system tends to develop an acceptable asymptotic behavior. In general, this paper shows the importance of simulation as a valuable tool for evaluating the overall quality of solutions provided by the MAS before the actual deployment.

6. REFERENCES

[1] Allen, J., Hendler J., and Tate A., (eds.). Readings in Planning, 1st ed., Morgan Kaufmann, 1990.

[2] Bernstein, P., Hadzilacos V., and Goodman, N. Concurrency Control And Recovery in Database Systems, Addison-Wesley, 1987.

[3] Carey, M.J., Franklin, M.J., Livny, M., and Shekita, E.J. Data Caching Tradeoffs in Client-Server DBMS Architectures, Proc. ACM SIGMOD, 357-366, 1991.

[4] Ferber, J. Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence, 1st ed., Addison-Wesley, 1999.

[5] FIPA ACL standard. http://www.fipa.org, 2000.

[6] Klusch, M. (ed.). Intelligent Information Agents: Agent-Based Information Discovery and Management on the Internet, 1st ed., Springer-Verlag, 1999.

[7] Nagi, K. Transactional Agents: A Robust Approach for Scheduling Orders in a Competitive Just-In-Time Manufacturing Environment. Proc. Workshop on MAS in logistics and economical perspectives of agent conceptualization, September 1999.

[8] Nagi, K. Scalability of a Transactional Infrastructure for Multi-Agent Systems. Proc. 1st Workshop on Infrastructure for Scalable Multi-Agent Systems at Autonomous Agents 2000, June 2000.

[9] Nagi, K., and Lockemann, P. Implementation Model for Agents with Layered Architecture in a Transactional Database Environment, Proc. 1st Workshop on Agent Oriented Information Systems, June 1999.

[10] Nodine, M. Interactions: Multidatabase Support for Planning Applications. Ph.D. Thesis, Brown University, 1993.

[11] Traiger, I.L., Trends in Systems Aspects of Database Management. Proc. 2nd International Conference on Databases, Wiley & Sons, 1983.

[12] Weiss, G. (ed.). Multi-Agent Systems: A Modern Approach to Distributed Artificial Intelligence, 1st ed., MIT Press, 1999.