A Load Sharing Approach Based on Refactoring of Roles in Multi-Agent Systems

Sebnem Bora, Ali Murat Tiryaki, and Oguz Dikenelli

Ege University, Department of Computer Engineering,
35100 Bornova, Izmir, Turkey
(sebnem.bora, ali.murat.tiryaki, oguz.dikenelli)@ege.edu.tr

Abstract. In this paper, we present a load sharing approach based on refactoring of roles of agents. According to our approach, a heavily loaded role is refactored by splitting it into new sub-roles with respect to a policy held by the "splitting ontology". This approach defines a new agent called "monitor agent" which monitors workload of roles of agents in the organization and decides on refactoring of roles. The monitor agent uses the platform ontology which explicitly describes components of the agent organization including agents, agents' roles, their plans, and their workload. This ontology is updated by the monitor agent in every monitor cycle.

1 Introduction

In distributed systems, computational elements work cooperatively so that large workloads should be allocated among them in an effective manner. Any strategy for workload distribution among computational elements is called load distribution. Load distributing algorithms can be classified as load balancing and load sharing. Load sharing is a distribution of processing and communication activities between computers so that no single computational element is overwhelmed. Load balancing algorithms attempt to equalize loads at all computational elements.

In a computer system, resource queue lengths and CPU queue lengths are good indicators of load, since they establish mutual relations with the task response time. If the load of a system is large, then the system suffers from performance degradation. The function of a load distributing algorithm is to transfer load from heavily loaded computers to lightly loaded computers.

Load distributing algorithms can be classified as static or dynamic. In static load distributing algorithms, decisions are taken using a priori knowledge of the system. In dynamic load distributing algorithms, we need to use the system's state information to reach a decision on distributing of loads at runtime.

As distributed systems, multi-agent systems (MAS) consist of autonomous agents that collaborate with each other to achieve a common goal. During the deployment phase, roles are assigned to agents on the verge of their execution. Roles are architectural elements which satisfy system goals collaboratively. Each

role has some responsibilities (agent goals), abilities (plans), authorizations and rules, all of which are based on system goals.

Due to openness of MAS, it is almost impossible to guess load of the agents in MAS during the analyze and design phases exactly. Therefore, load of the agents that play a specific role in MAS may increase at run time unexpectedly. If the total request volume at any time is unusually high on an agent, then this may lead to the agent failing with all requests unable to be performed.

Agent is only container used to execute developed roles like objects in traditional object oriented development - OOD. All features of an agent come from the roles that are assigned to it. Therefore, overloading of an agent also comes from its role(s). In such a condition, the overloaded entity is not agent itself, the role(s) played by failing agent(s) is overloaded. Assigning these roles to the other agents in the organization does not solve overloading, since the overloading is passed to these agents via the overloaded role(s) assigned to them. To solve overloading, the role(s) have to be split. In this case, transferring some plans of a role which cause heavy workloads, to recent created roles potentially improves system's performance.

In this paper, we present a load sharing approach based on applying refactoring [5] practice of extreme programming [1] on roles. Responsibilities and abilities of roles may change in dynamic and open MASs frequently. Hence, the concept of role is one of the most critical elements for refactoring in such systems. At this level, refactoring techniques such as moving responsibilities related with the role are needed to improve the role structure of MAS at hand. Several roles cooperate to achieve a common system goal of the system being developed. Since refactoring techniques applied on the roles do not change external behaviors of the roles, achievement of system goals is not affected from the refactoring operations.

According to our approach, the role on the heavily loaded agent is divided into new sub-roles with respect to a policy held by the "platform ontology". "Platform ontology" is a general ontology to model multi-agent platform from load sharing perspectives. It explicitly describes components of the platform and agents, their initial values, and the load sharing policy. The strategy used in our approach is to distribute the plans of a role that cause having large workload, to new roles defined in design phase and described by the rules of the "platform ontology". The process of splitting a role to new roles at runtime is called role refactoring. This approach uses a mechanism that monitors workloads of the agents in the organization and decides on refactoring of roles of agents.

In order to implement load sharing technique based on applying role refactoring, we have investigated several works. The static semantics of roles, formation of roles, interactions between roles have been investigated in [2,7,10,3].

In traditional object-oriented design, roles are implicit to objects that play them and there is no dynamic adaptation of the structure of the relationships between objects and roles in response to changes in environment, user or application requirements, and available resources. Colman and Han propose an object-oriented role-based methodology based on the concept of ontogenic adap-

tation to make a system more adaptable [Colman05]. Ontogenic adaptation requires interchangeable elements, flexible structure, and organizational regulation. These all can be achieved through creating decoupled object-role structures. This methodology distinguishes three types role in an organization: functional roles, operational-management roles and organizational management roles. Functional roles define the processes of the system that achieve the system's functional requirements. Operational management focuses on regulating the relationships between functional roles. Organizational management provides mechanisms to ensure their systems' viability by both ensuring the internal integrity of the system and monitoring the external system performance, and to effectively modify their organizational structures. The binding of those roles to objects creates on instantiation of the role-based organizational structure.When the system needs to change its organization to meet the new goals or to adjust environmental requirements, the organizational-management roles manage the re-allocation of roles and associations.

The work on formalizing dynamic role assignment has been done by Odell et. al [8]. In their work, they discuss the "role-changing" operations by partitioning into categories of dynamics: dynamic classification and dynamic activation. The concept of dynamic activation describes how an agent activates or suspends the various roles that it plays. The concept of dynamic classification conceives of roles being bound to an agent after the agent is deployed, and unbound without terminating the agent.

The main difference of our work from others is to dynamically refactor roles of agents to distribute large workloads on heavily loaded agents at runtime. To our knowledge, our load sharing approach is the first one that uses refactoring practice for managing workload of the agents in the organization (not only in agent systems but in general). Moreover, since the refactoring policy is defined using a pre-known ontology and stored by the agent organization, changing of this ontology at runtime makes the agent organization adapt itself to a new load sharing policy.

The remainder of this paper is structured as follows: Section 2 presents the context of the work; Section 3 introduces an abstract architecture for load sharing in MAS organizations; Section 4 briefly describes how to acquire information to determine the excess load of agents in our load sharing approach and presents how to achieve load sharing in MAS; Section 5 gives the evaluation of the approach; and finally Section 6 gives the conclusion.

2 Realization of Load Sharing Components in a MAS Environment

Dynamic load distributing algorithms (load sharing and load balancing algorithms) outperform static load distributing algorithms for distributed systems, since they are able to exploit changes in the system state. This characteristic of dynamic algorithms is also critical for MAS organizations, since workload of

agents may change unexpectedly due to openness of MAS organizations. Therefore, load distributing approaches in MAS must use dynamic algorithms.

Typically, a dynamic load distributing algorithm has four components: (1) a transfer policy, (2) a selection policy, (3) a location policy, and (4) an information policy.

Transfer policies determine whether an object may participate in a task transfer. They define threshold policies. Thresholds are expressed in units of workload. When the workload on the object exceeds a threshold T, transfer policies decide that the object is a sender. If the workload falls below T, the transfer policy decides that the object is a receiver. When the transfer policy decides that the object is a sender, a selection policy selects a task for transfer. The simplest approach is to select tasks that have caused the node to become a sender by increasing the workload. A location policy finds suitable senders or receivers to share load. It is usually achieved through polling. Finally, the information policy decides when information about the other objects in the system should be collected, where it should be collected from, and what information should be collected. [11].

In this section, we briefly describe our approach for dynamic load sharing in MAS organizations. To define such an approach, one has to identify a selection policy first. We use the MAS meta model shown in Figure 1 to define our selection policy. There are many agent system development methodologies such as [13], [6] and different meta models that have been proposed by these methodologies in the literature. However, we defined a generic meta model that can be used any methodology. To apply our load sharing approach in MAS, we added workload concept to our MAS meta model. This concept includes data about the current state of each role.

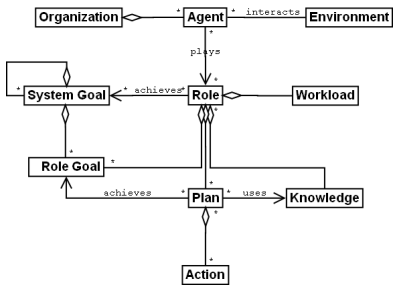

Fig. 1. MAS meta model

Now, it is time to define the selection policy based on the defined meta model. Since the selection policy selects the tasks that cause the overloading, we then have to answer the basic question, "what makes an agent overloaded?". Like

any computational entities, an agent is overloaded due to tasks processing and messaging. Since these tasks and messages belong to the role(s) of the agent, we can conclude that overloading is dependent to the role(s). Although one can identify the overloaded role, this role cannot be selected as a distributed task since the transfer of role makes another agent overloaded again. So, an agent has to split a role to sub-roles to distribute its workload. Dividing the roles to sub-roles creates a role model where each sub-role takes some responsibilities of the original role.

The general methodology of creating such a role model is to split the original role to coherent roles that semantically encapsulate the related behaviors of the original role. Of course, such a modeling must have been done by designers of the organization and explicitly represented to be used by the agent(s) to apply the selection policy. This splitting role model can be represented by a simple ontology. This ontology that is defined by the designers at the design holds the roles that can be overloaded at the run-time and the role splitting strategies that are used to split these overloaded roles. The splitting strategy includes sub-roles of the role and plans of each sub-roles. The structure of this ontology is shown in Figure 2.

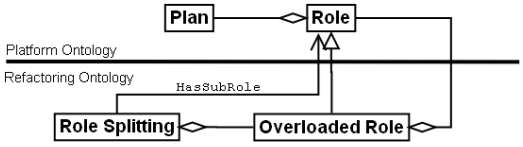

Fig. 2. The Meta Model for Splitting Role

The selection policy simply selects a sub-role to transfer by considering the workload of the role. Although it is possible to dynamically calculate the workload of a role, this is not a straight-forward process. Because, each role includes plans and each plan may interact with other agents and creates a messaging load. So, one needs a complex monitoring mechanism to observe and calculate load of each role of the agents. In this paper, we have implemented a monitoring mechanism that observes the role's workload. Thus our selection policy simply selects the predefined sub-role to be transferred based on the workload of the role and a refactoring ontology.

The second policy for load sharing is the location policy which is used for finding a suitable receiver that can overcome excess loads. From MAS perspective, the location policy has to identify the suitable agent to transfer the selected sub-role. The location policy associates with the monitoring mechanism to identify the suitable receiver agent since the monitoring mechanism holds the current

states of the agents that exist in the organization. Hence, it computes the excess workload of each agent by comparing the agent's workload to the average over all load of the system. The agents with less load are identified as the receivers or a new agent is created as a receiver in a suitable ground machine.

In order to determine the load of each role and the workload of each agent, we have to monitor the environment to collect information. The information policy decides when information about the agents in the system should be collected, where it should be collected from, and what information should be collected. Our approach uses a periodic and centralized information policy, in which each domain agent periodically sends its state to the monitoring mechanism.

In the monitoring mechanism, the excess load of agents and roles are evaluated by applying our threshold policy. If the workload caused by a specific role is larger than the average load in the organization, the monitoring mechanism decides on applying the role refactoring algorithm on a role. The threshold policy will be detailed in Section 4.1.

3 The Abstract Architecture

The proposed abstract architecture is illustrated in Figure 3 and built on the FIPA (Foundations for Physical Agents) based MAS architecture[1]. To collect the workload related data and evaluate the collected data, we propose a specific role called "monitor role". The agent that plays the "monitor role" is called "monitor agent".

In this abstract architecture, the monitor agent is a centralized agent that controls domain agents in the organization at runtime. It holds the current state of the platform in an OWL[2] ontology called platform ontology. The monitor agent receives workload messages sent by domain agents and pass the data extracted from those messages to a plan called the "workload evaluation plan" of the "monitor role". This plan evaluates the load of each role in the organization. It also evaluates the current state of each agent that sends its workload data within the content of the message, according to the current state of other agents in the platform using the platform ontology. If the monitor agent detects that an excess load for a role exists during this plan execution, it then sends a message to itself to activate the "role splitting plan". This plan decreases the responsibilities of the role by splitting the role to sub-roles using the load sharing guidelines defined at the design time. The main approach for role splitting is to split plans of the role to new roles.

[1] FIPA, FIPA Specifications , http://www.fipa.org
[2] Web Ontology Language (OWL), http://www.w3.org/2001/sw/WebOnt/

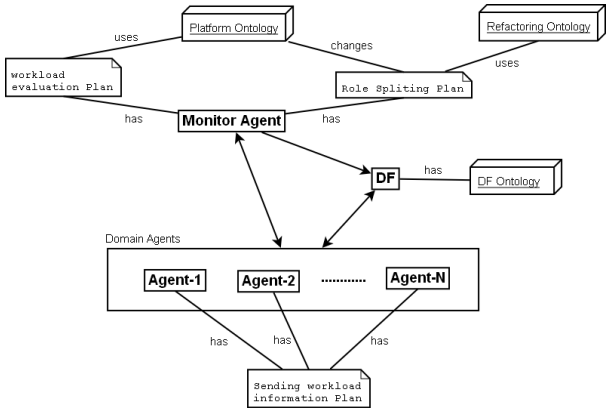

Fig. 3. Abstract architecture for load sharing based on refactoring

As aforementioned, data related to the excess load for each role is acquired directly from the domain agents. Domain agents are agents which fulfill requirements of domain. Each of these agents has the plan library based on their responsibilities. Naturally, the agent's responsibility comes from the roles that agent has to play. All of the domain agents in the MAS may execute more than one role to achieve general goals of the organization. Roles are identified during the design of the organizational structure and assigned to the agent when the agent is initiated. Here, each domain agent executes a plan named "Send workload to monitor agent". This plan is periodically executed during the agent's operation. In this plan, firstly the workload data of the role played by the agent is collected from the agent infrastructure in a certain period. The collected data consists of the agent's roles, the number of objectives (goals extracted from incoming requests) for each role of the agent (the workload of the role), the mean of the time for each agent to perform requests, and arrival rates of requests. Finally, the collected data is sent within the content of the inform message to the monitor agent. Next, we explain the components of our load sharing approach based on role refactoring in detail.

4 Monitoring for Load Sharing in MAS

A monitor agent is a centralized and critical agent that plays the "monitor role". It continuously monitors a multi-agent organization and ensures reliability of the MAS by refactoring on the organizational structure when it detects an abnormal state such as overloading of agents in the organization. The monitor agent has two critical responsibilities: evaluation of the collected information and applying of role splitting when an overloaded role is identified. In order to achieve its responsibilities, first it has to evaluate workloads of the agents in the organization

to catch system's abnormal states; then it has to split the role when it catches an overloading on a role. In the following subsections, we explain in detail how the monitor agent fulfills these two responsibilities.

4.1 Evaluation of the Collected Information

It is time to explain how we process data collected by the monitor agent to obtain the excess load information of each agent and each role in the organization. In order to explain how we process data, first we need to explain our transfer policy. It is very similar to the transfer policy proposed by Dhakal et. al in [4]. It uses a threshold policy which defines the excess workload of a role. This threshold policy is based on a queuing model which characterizes the stochastic dynamics of the load distributing in a multi-agent organization of n agents which collaborate with each other.

In this architecture, clients (agents or human users) send "request" messages to agents to perform some actions. When an agent receives a "request" message, it matches the goal extracted from the incoming "request" message to an agent plan. Then, this plan is scheduled and executed by the agent's internal architecture.

We assume that in this agent system, requests arrive according to a Poisson process of rate λ so the interarrival times are i.i.d (independent, identically distributed) exponential variables with mean $1/\lambda$[9]. Also, we consider that each agent performs requests according to an exponential distribution with mean $1/\mu$ and sends its workload information to the monitor agent at every period.

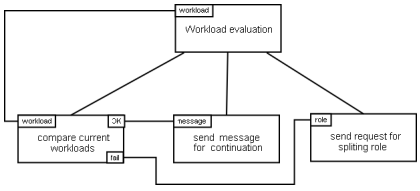

Fig. 4. The "workload evaluation" plan of the monitor agent

When the monitor agent receives a workload message from the agent, it initiates the "workload evaluation" plan. The structure of this plan built by using the hierarchical task network (HTN) formalism [12] is shown in Figure 4. According to this plan, the workload data obtained from the workload message is transferred to the "compare current workloads" primitive task by an inheritance link. In this primitive task, the monitor agent computes the excess workload of each role by comparing the role's workload to the average over all load of the system.

During system initialization, a period is set for the organization. This period is called the sampling period T and actually defined over a time window ((k-1)T, kT), where k is the sampling instant. The excess workload of each role is given by the following equation:

$$L_i(k) = \sum_{j=0}^{n} Q_{ij}(k) - \frac{\sum_{j=0}^{n} 1/\mu_{ij}}{\sum_{h=0}^{m}\sum_{j=0}^{n} 1/\mu_{hj}} \sum_{h=0}^{m}\sum_{j=0}^{n} Q_{hj}(k) \quad (1)$$

$\boldsymbol{L_i(k)}$**:** The excess load of the ith role at the kth sampling period .

$\boldsymbol{Q_{ij}(k)}$**:** The total workload of the ith role at the kth sampling period. Since there are n agents in the organization, the monitor agent determines the total workload of the ith role by considering the workload data sent by every agent.

μ_{ij}**:** The j^{th} agent performs requests for the ith role according to an exponential distribution with mean μ_i. In order to obtain the workload of the ith role, the monitor agent determines the sum of all the $(1/\mu_i)$s received from all the agents that play the ith role.

$\boldsymbol{Q_{hj}(k)}$**:** The workload data of the hth role at the kth sampling period. The monitor agent determines the total workload of the roles of n agents in the organization.

μ_{hj}**:** The jth agent performs requests for the hth role according to an exponential distribution with mean μ_h.

The second quantity in equation 1 is the fair share of the ith role from the total workloads in the system. The excess load of the ith role is restricted by the following:

$$L_i(k) > a\frac{\frac{1}{n}\sum_{j=0}^{n} \mu_{ij}}{\lambda_i} \quad (2)$$

n: The number of agents in the organization.

λ_i**:** The arrival rate of requests for the ith role.

a: The coefficient which is defined by the programmer. a is selected as 0.3 in this work.

If the monitor agent detects that the excess load of the ith role is greater than the right-hand side of the inequality 2, it then decides that the ith role is a sender. According to the "workload evaluation" plan, the "compare current workloads" task returns the "fail" outcome with respect to the overloaded role. The overloaded role definition is admitted by the "send request for role splitting" primitive task. In this task, the monitor agent prepares the message that includes the request for splitting of the overloaded role and sends this message to itself to initiate the

related plan. If there is no excess load condition for the agent, then "compare current workloads" primitive task produces "OK" outcome.

In this section, we mentioned about our transfer policy for load sharing based on refactoring of the roles of heavily loaded agents. In the next section, we explain our selection policy that selects the roles that have caused the agent to become a sender and our location policy.

4.2 Splitting Roles

When the monitor agent receives the message that includes the request for role splitting, it initiates the"role splitting" plan shown in Figure 5.

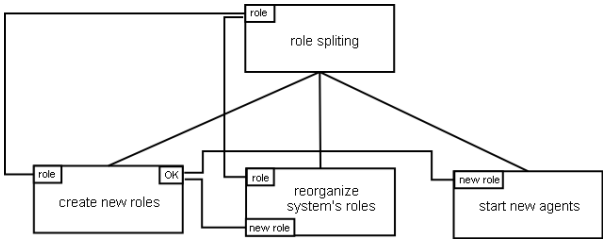

Fig. 5. The role splitting" plan

This plan takes the role that makes the agent(s) overloaded. To split the role, first, the sub-roles of the overloaded role and the plans that would be executed by these roles have to be determined. Currently, we consider the information about how overloaded role is split into sub-roles at the design time. Developers determine the splitting strategy and save this information as an instance of the role splitting concept in the refactoring ontology. In the "create new roles" task, this splitting strategy for the overloaded role is obtained from the refactoring ontology with respect to the received "role" provision. Then, the platform ontology of the system is changed by adding new roles defined as the sub-roles of the original role in the role splitting strategy. At this point, the new role instances are created in the ontology. However, there is not any agent that plays these roles in the platform. The definitions of the new roles are passed to the "reorganizing systems' roles" and "start a new agent" complex tasks. The HTN structure of the complex task called "reorganize system's roles" is shown in Figure 6.

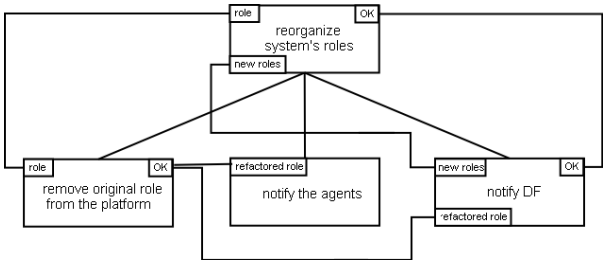

Fig. 6. The "Reorganize system's roles" plan of the monitor agent

In the previous task, the new role with their plans was created. However, the overloaded role were not removed from the platform ontology and the domain agents that play original roles still use the plan library that is not relieved. In the first primitive task, the overloaded role is removed from the platform ontology. In the next task, a message that notifies removing of the overloaded role is sent to the agents that play the overloaded role.

Since a role is removed from the platform and some new roles are added to the platform, the distribution of roles' responsibilities and capabilities in the MAS are changed. At this point, these changes have to be notified to the directory facilitator (DF) of the platform. In the primitive task called "notify DF", a notification message that defines the removing role, the new roles and their capabilities is prepared and sent to the directory facilitator.

The directory facilitator stores two ontologies in its knowledge base. One of these ontologies includes services that are supplied by the roles as service descriptions defined by FIPA. The other ontology includes the information about the roles that the agent plays in the platform. So, the mentioned notification message causes that the service description ontology is changed by DF.

According to our location policy, if the agents with less load can not be identified as receivers, then new agents are created as receivers in suitable ground machines by executing the final complex task called "start new agents" . In this complex task, first, the ground machines are selected from available machines submitted to the platform in accordance with our location policy. During this selection process, the monitor agent looks up some workload values -like CPU load- of the domain agents and selects the most suitable machine that can overcome more loads based on the information collected before. After selecting the suitable machines, the plan library of the new role is sent to these machines. Finally, an agent that uses this plan library is started on each of these selected ground machines in the"start the agents" task. Also, the notification messages include advertisements of the started agents and their roles.

5 Evaluation of the Approach

Our load sharing approach presented in this paper has been implemented with the SEAGENT framework [3].

5.1 Modeling Perspective for Applying Load Sharing

To show the effectiveness of our load sharing approach, we applied it to an agent system application which was implemented by SEAGENT Research Group. The case focuses on one of the core scenarios of the tourism domain. In this scenario, the traveler tries to organize a holiday plan which includes hotel booking and transportation details. It is assumed that accommodation and transportation preferences of the traveler are known by the system. The aim of the scenario is to arrange the cheapest holiday plan by selecting the proper accommodation and transportation options based on the traveler preferences. The primary roles of the scenario are identified as the "Traveler", "travel agency", "Hotel", and "Transportation Provider".

In the initial design of the system, it has been realized that the travel agency role is the most critical role in terms of the load sharing perspective, because the agent that plays this role has two critical responsibilities which are hotel booking and proper transportation selection. The role diagram of our initial design is shown in Figure 7.

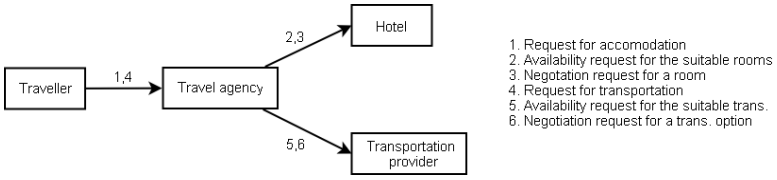

Fig. 7. initial role diagram of the "prepare a holiday" scenario

After we realized that the travel agency role can be heavily loaded, we applied our load sharing at the design phase. Based on our approach, we defined a role splitting strategy for developing a tourism system. This strategy is defined as a role splitting individual, which is stored in the monitor agent's knowledge base. This individual is shown in Figure 8.

[3] Semantic Web Enabled Multi-Agent Framework, SEAGENT, http://www.seagent.ege.edu.tr

```
<Role_Splitting rdf:ID="Role_Spilitting_for_Tuorism">
  <hasOverloaded_Role>
    <Overloaded_Role rdf:ID="Travel_Agency">
      <hasSubRole>
        <Platform:Domain_Role rdf:ID="Transportation_Agency">
          <Platform:HasPlan rdf:resource="#Arrange_Trasportation"/>
        </Platform:Domain_Role>
      </hasSubRole>
      <hasSubRole>
        <Platform:Domain_Role rdf:ID="Hotel_Agency">
          <Platform:HasPlan rdf:resource="#Arrange_Accomodation"/>
        </Platform:Domain_Role>
      </hasSubRole>
    </Overloaded_Role>
  </hasOverloaded_Role>
</Role_Splitting>
```

Fig. 8. The role splitting individual for the "travel agency" role.

This Role_Splitting individual includes the Overloaded_Role individual that is referenced with "Travel_Agency". This individual defines what should be done, when the "Travel_Agency" is overloaded. According to this strategy, the original travel agency role is split into two role concepts which are hotel agency and transportation agency roles. So, these two new domain roles have been added to the definition of the "Travel_Agency" overloaded role individual. The "hasPlan" property of the "Hotel_Agency" role was set with the reference of one of the "Travel_Agency" role's plans called "Arrange_Accommodation". Likely, the same property of the other sub-role called "Transportation_Agency" was set with the reference of the other plan called "Arrange_Transportation" of the overloaded role. So, this role splitting individual specifies when an overloading is occurred on the travel agency role, when two new roles are created, and when one of the plans of original roles is assigned to each of these new roles.

The monitor agent which manages the splitting strategy is initialized with its plans , if the travel agency role is overloaded. During the system execution, we observed that the travel agency role is split into two sub-roles defined in the design phase, when the number of the travelers in the system increased considerably.

At the end of refactoring, we observed that workload of the travel agency reduced remarkably. So, the overloading problem on the travel agency role has been solved by using our refactoring based load sharing approach. The role diagram for the travel agency role is shown in Figure 9.

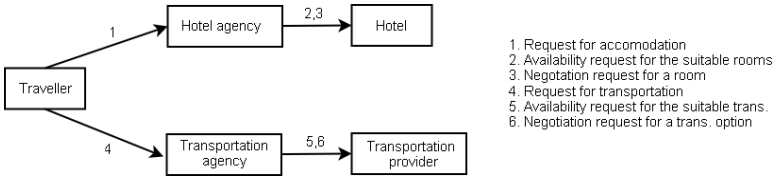

Fig. 9. The role diagram based on the defined role splitting strategy for travel agency role

5.2 Discussion

Figure 10 illustrates the performance results of the implementation. To get these performance results, the traveler role is assigned to an agent which executes simple plans to send hotel booking and transportation requests equally to the travel agency role using an exponential distribution. The travel agency role is executed by another agent which has two separate plans for hotel booking and transportation selection. In the first case, while the system does not apply any load sharing technique, we observe the response time of the system, as the number of requests per 10 seconds sent to the travel agency increases. In the second case, while the system applies the load sharing technique based on refactoring of roles of agents, we observe the response time of the system as the number of requests per 10 seconds sent to the travel agency increases. In both cases, agents report their workload data to the monitor agent at every 10 seconds. When the load sharing setting is deployed, the interaction diagram is shown in Figure 11.

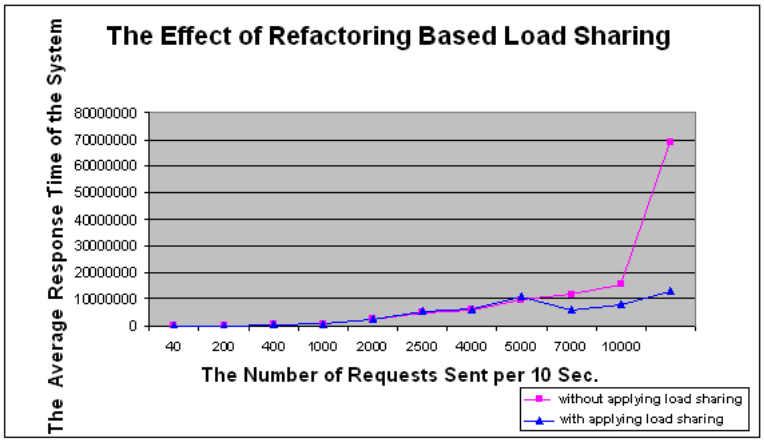

Fig. 10. The Effectiveness of the Load Sharing Approach

As seen from the Figure 10, the response time of the system without applying load sharing increases, as the number of the requests per 10 seconds sent to the travel agency increases. This result was expected, since the travel agency has to respond to more requests. In the second test, the monitor agent detects that the travel agency becomes a sender at a certain point. It then decides on splitting the travel agency's plans into two different roles as described in the splitting ontology. It also starts a new agent on another computer and sends one of the roles to it while reorganizing the other role. As seen from the figure, the system applying the load sharing approach shows better performance compared to the system without applying load sharing, since we share the system's load between agents that play each sub-role.

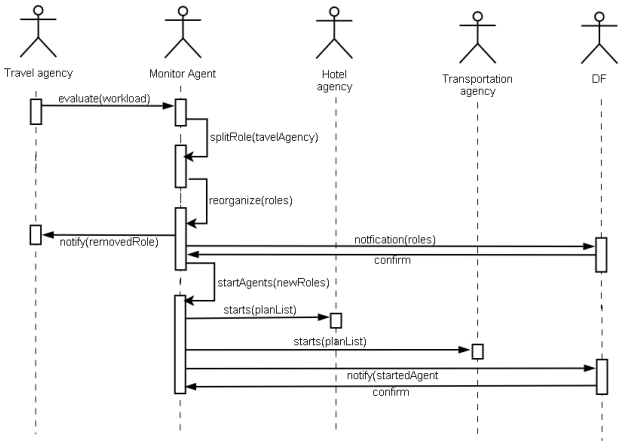

Fig. 11. Interaction diagram for role splitting process

6 Conclusion

In this paper, we present a load sharing approach based on refactoring of roles that have caused agents heavily loaded. In this approach, we identify the roles that have caused the agent to be overloaded, and then decrease the responsibilities of these roles by splitting them into sub-roles using the load sharing guidelines defined at the design time.

In order to identify the roles that are to be split, we need a complex monitoring mechanism to observe and calculate load of each role of the agents. Therefore,

we have implemented a monitor agent which continuously monitors a multi-agent organization.

While evaluating the proposed architecture, we observe that the load sharing approach based on refactoring of roles improves system's performance under heavy load conditions. In conclusion, first results are encouraging and indicating that efficient load sharing in multi-agent organizations is sustainable using this model.

References

1. Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, 1999.
2. Cristiano Castelfranchi. Engineering social order. In *Engineering Societies in the Agents World*, volume 1972 of *LNAI*, pages 1–18. Springer-Verlag, December 2000. 1st International Workshop (ESAW'00), Berlin (Germany), 21 August 2000, Revised Papers.
3. Mehdi Dastani, Virginia Dignum, and Frank Dignum. Role-assignment in open agent societies. In *AAMAS '03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 489–496, New York, NY, USA, 2003. ACM.
4. Sagar Dhakal, Jorge E. Pezoa, and Cundong Yang. Dynamic load balancing in distributed systems in the presence of delays: A regeneration-theory approach. *IEEE Trans. Parallel Distrib. Syst.*, 18(4):485–497, 2007. Senior Member-Majeed M. Hayat and Senior Member-David A. Bader.
5. Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.
6. F. Giunchiglia, J. Mylopoulos, and A. Perini. The tropos software development methodology: Processes, 2001.
7. James Odell, H. Van Dyke Parunak, and Bernhard Bauer. Representing agent interaction protocols in UML. In *AOSE*, pages 121–140, 2000.
8. James Odell, H. Van Dyke Parunak, Sven Brueckner, and John A. Sauter. Changing roles: Dynamic role assignment. *Journal of Object Technology*, 2(5):77–86, 2003.
9. A. Papoulis. *Probability, Random Variables, and Stochastic Processes*. Mc-Graw Hill, 1984.
10. H. Van Dyke Parunak and James Odell. Representing social structures in UML. In Jörg P. Müller, Elisabeth Andre, Sandip Sen, and Claude Frasson, editors, *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 100–101, Montreal, Canada, 2001. ACM Press.
11. Mukesh Singhal, Niranjan G. Shivaratri, and Niranjan Shivaratro. *Advanced Concepts in Operating Systems*. McGraw-Hill, Inc., New York, NY, USA, 1994.
12. M. Williamson, K. Decker, and K. Sycara. Unified information and control flow in hierarchical task networks. In *Theories of Action, Planning, and Robot Control: Bridging the Gap: Proceedings of the 1996 AAAI Workshop*, pages 142–150, Menlo Park, California, 1996. AAAI Press.
13. Franco Zambonelli, Nicholas R. Jennings, and Michael Wooldridge. Developing multiagent systems: The gaia methodology. *ACM Trans. Softw. Eng. Methodol.*, 12(3):317–370, 2003.