Turk J Elec Eng & Comp Sci, Vol.20, No.2, 2012, © TÜBİTAK
doi:10.3906/elk-1101-1044

Load sharing based on moving roles in multiagent systems

Şebnem BORA[1,*], **Ali Murat TİRYAKİ**[2], **Oğuz DİKENELLİ**[1]
[1] *Department of Computer Engineering, Ege University, İzmir-TURKEY*
[2] *Department of Computer Engineering, Çanakkale Onsekiz Mart University, Çanakkale-TURKEY*
e-mail: sebnem.bora@ege.edu.tr

Received: 23.01.2011

Abstract

In this paper, we present a load-sharing approach based on the refactoring of agents. According to our approach, the role(s) that makes an agent overloaded is identified and transferred to less loaded agents. The excess workload of this heavily loaded agent is then transferred to the new agent. This approach defines a new agent, called the "monitor agent," which monitors the workload of agents in the organization and decides about the refactoring of the agents. The monitor agent uses the platform ontology, which explicitly describes the components of the agent organization, including agents and their roles, plans, and workloads. This ontology is updated by the monitor agent in every monitor cycle.

Key Words: *Distributed computing, dynamic load sharing, multiagent systems, software agents*

1. Introduction

As distributed systems, multiagent systems (MAS) consist of autonomous agents that collaborate with each other to achieve a common goal. Due to the openness of MAS, it is almost impossible to accurately guess the load of the agents in MAS during the analysis and design phases. Therefore, the load of the agents in MAS may unexpectedly increase at runtime. If the total request volume at any time is unusually high on an agent, this often leads to the agent failing, with all requests unable to be performed. In this case, transferring the workload of an agent that has heavy workloads to idle or lightly loaded agents potentially improves the system's performance.

In this paper, we present a load-sharing approach based on the refactoring of agents. According to our approach, the agent is the container used to execute developed roles, like objects in traditional object-oriented development. During the deployment phase, roles are assigned to agents on the verge of their execution. Roles are architectural elements that satisfy system goals collaboratively. Each role has some responsibilities (agent goals), abilities (plans), authorizations, and rules, all of which are based on system goals. All features of an

[*]Corresponding author: Department of Computer Engineering, Ege University, İzmir-TURKEY

agent come from the roles that are assigned to it. Therefore, the overloading of an agent also comes from its role(s). In such a condition, the failing agent(s) is overloaded because of the roles that it plays. Assigning the role(s) to lightly loaded agents in the organization and transferring the excess loads can solve the overloading problem.

According to our approach, the tasks of the heavily loaded agent are distributed with respect to a policy held by the "platform ontology." "Platform ontology" is a general ontology to model a multiagent 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 transfer an agent's role that causes a large workload to another agent. The agent with the large workload can then transfer its excess tasks to that other agent. The process of moving a role to a new agent at runtime is called "agent refactoring" [1,2]. This approach uses a mechanism that monitors the workloads of the agents in the organization and decides to apply the load-sharing algorithm to transfer loads from heavily loaded agents to lightly loaded agents.

In order to implement the load-sharing technique in our work, we investigated several architectures that apply load-distributing policies. Schaerf et al. [3] studied the process of multiagent reinforcement learning in the context of load balancing in a distributed system without the use of central coordination or communication. In their work, the information an agent gets is purely local. The agent will not be informed by the other agents how efficient the service that it requests is or that how large their workloads are.

Appleby and Steward [4] proposed the use of larger numbers of mobile software agents to improve load balancing. In their approach, there are 2 types of mobile agents: load management agents and parent agents. The lowest level of control is provided by the load management agents. They move around the network and find the best routes from all nodes in the network to the source node. Parent agents travel over the network, gathering information about which nodes are generating the most traffic and which nodes are more congested than others. A parent agent then decides that network management at certain locations is needed to relieve congestion.

In [5], Cao et al. focus on grid load-balancing mechanisms using agents. Each agent is responsible for resource scheduling and load balancing across multiple hosts in a local grid. The agent utilizes application performance data with iterative heuristic algorithms to dynamically minimize the workload. At a higher level, agents cooperate with each other to balance the workload using a peer-to-peer service advertisement and discovery mechanism.

Work on distributed computer systems usually adopts the view of a set of computers, each of which controls certain resources and workloads that arrive to it in a dynamic fashion [6]. The decision-making components of such systems share the system's load by means of communication. In our work, we also take advantage of the communication between agents. However, the main difference between our work and others is that our load-sharing approach is based on agent refactoring.

The remainder of this paper is structured as follows: Section 2 presents the context of the work and the realization of load-distributing components in a MAS environment, Section 3 presents the abstract architecture that we propose and the monitoring for load sharing in MAS organizations, Section 4 gives the evaluation of the approach, and Section 5 gives the conclusion.

2. Context of this work

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 loads from heavily loaded computers to lightly loaded computers.

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

Dynamic load-distributing algorithms outperform static load-distributing algorithms, since they are able to exploit changes in the system state. However, dynamic load-distributing algorithms entail overhead in the collection, storage, and analysis of system state information.

A large number of distributing algorithms have been proposed in distributed systems. Typically, a load-distributing algorithm has 4 components: a transfer policy, a selection policy, a location policy, and an information policy.

Transfer policies determine whether a node may participate in a task transfer. They define threshold policies. Thresholds are expressed in units of workload. When the workload on a node exceeds a threshold, some transfer policies decide that the node is a sender. If the workload falls below the threshold, the transfer policy decides that the node is a receiver.

A selection policy selects a task for transfer when the transfer policy decides that the node is a sender. 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 the load. The finding of a suitable node is achieved through polling. In polling, a node polls another node to find out whether it is a suitable node for sharing.

The information policy decides when information about the other nodes in the system should be collected, where it should be collected from, and what information should be collected. The next section describes the realization of load-distributing components in a MAS environment.

2.1. Realization of load-distributing components in a MAS environment

In this section, we briefly describe our approach for dynamic load sharing in MAS. To define such an approach, one has to identify a selection policy first. We used the MAS metamodel shown in Figure 1 to define our selection policy.

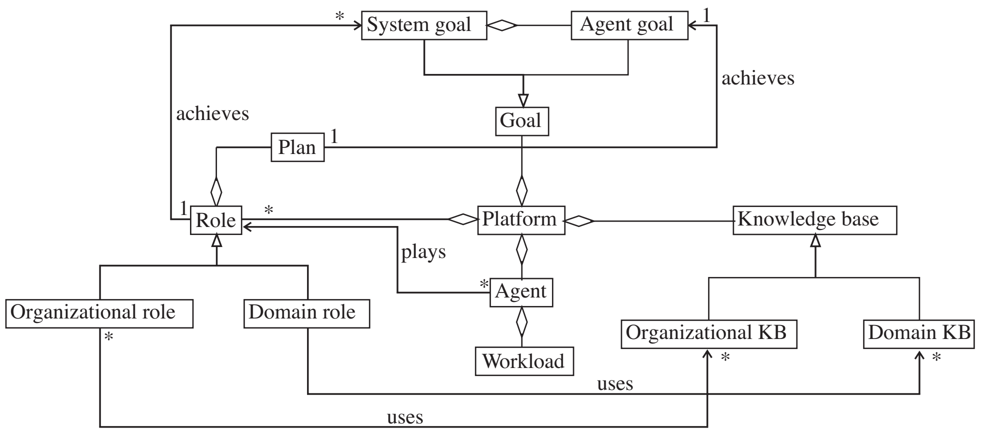

Figure 1. MAS metamodel.

There are many agent system development methodologies, such as those in [7,8], and different metamodels have been proposed according to these methodologies in the literature. However, we defined a generic metamodel that can be used with any methodology. According to our approach, agent systems are built to achieve some general goals called system goals. Each of the system goals is assigned a role. There are 2 kinds of roles in our metamodel. Organizational roles have some privileges, and the agents that play these roles can access organizational knowledge bases. Domain roles are executed by domain agents, which have some domain-dependent responsibilities. That kind of agent can only access the domain knowledge bases. Roles are assigned to agents to be executed. Agents achieve the goals of their own roles by executing their plans. System goals can be achieved by agents that cooperate with each other. In this cooperation, each agent has its own agent goals based on the roles it plays. An agent goal can be achieved by a plan. Organizational knowledge bases store information about the current state of the MAS organization. Domain knowledge bases are used to store domain-dependent information. To apply our load-sharing approach in MAS, we added the workload concept to our MAS metamodel. This concept includes data about the current state of each agent.

We can now define the selection policy based on the defined metamodel. Since the selection policy selects the tasks that cause the overloading, we then have to answer the basic question of what makes an agent become overloaded. Like any computational entity, an agent is overloaded due to task processing and messaging. Since these tasks and messages belong to the role(s) of the agent, we can conclude that overloading is dependent on the role(s).

The selection policy simply selects tasks to transfer by considering the workload of the agent. However, selected tasks are categorized based on roles that make the agent overloaded, since the selected tasks need to be processed by an agent that plays the same specific role. Although it is possible to dynamically calculate the workload of an agent, this is not a straightforward process. One needs a complex monitoring mechanism to observe and calculate the load of each agent. We therefore implemented a monitoring mechanism that observes the agent's workload based on its roles. Agents can overload because of their roles. Thus, our selection policy simply selects the tasks to be transferred based on the workload of the agent and the role played by that specific agent.

The second policy for load sharing is the location policy, which is used for finding a suitable receiver that can overcome excess loads. From the MAS perspective, the location policy has to identify a suitable agent for transferring the selected tasks. The location policy associates with the monitoring mechanism to identify a 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 overall load of the system. The agents with lighter loads are identified as the receivers, or a new agent is created as a receiver in a suitable ground machine.

In order to determine 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 is evaluated by applying our threshold policy. The threshold policy will be detailed in Section 3.

3. Monitoring for load sharing in MAS

The proposed abstract architecture for load sharing in MAS is illustrated in Figure 2 and was built on MAS architecture based on the specifications of the Foundation for Intelligent Physical Agents (FIPA) [9]. To collect and evaluate the workload-related data, we propose a specific role called the "monitor" role. The agent that plays the "monitor" role is called the 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 a web ontology language [10] called platform ontology. The monitor agent receives workload messages sent by domain agents and passes 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 agent based on their roles in the organization. If the monitor agent detects that an excess load for an agent exists during this plan's execution, it then sends a message to itself to activate the "role moving" plan. This plan decreases the task processing of the agent by using the load-sharing approach. As previously mentioned, data related to the excess load for each agent are acquired directly from the domain agents. Domain agents are agents that fulfill requirements of the domain. Each of these agents has a 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 MAS may execute more than one role to achieve the 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.

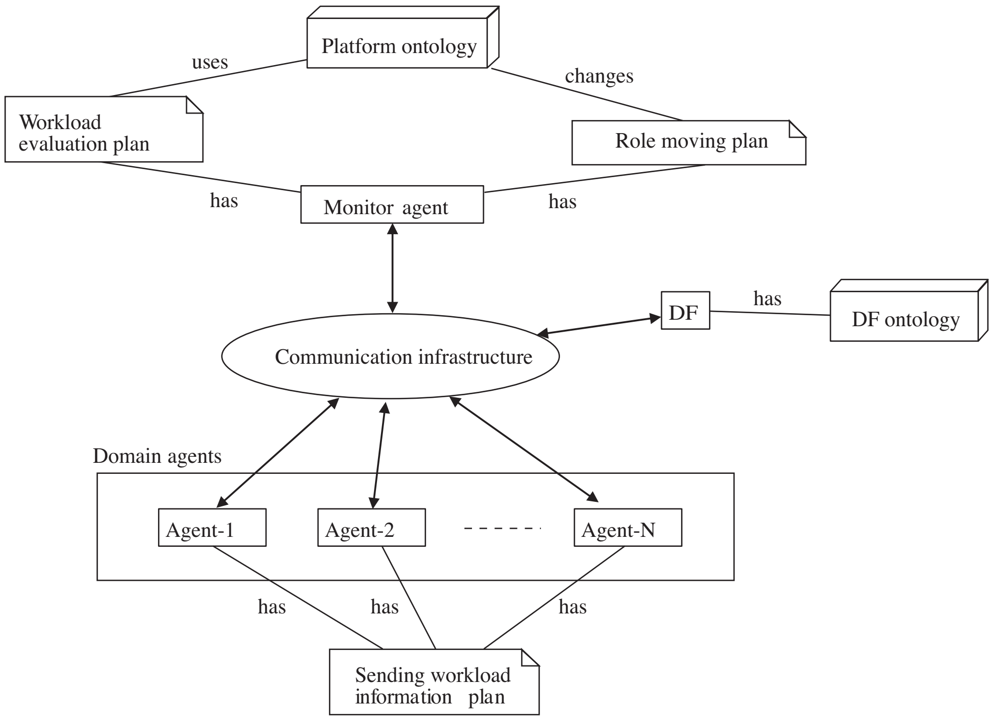

Figure 2. The abstract architecture.

Here, each domain agent executes a plan named "send workload" to the monitor agent. This plan is periodically executed during the agent's operation. In this plan, the agent's workload data are first collected from the agent's infrastructure within a certain period. The collected data consist 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 time for each agent to perform requests, and the arrival rates of requests. Finally, the collected data are sent within the content of the "inform" message to the monitor agent. Next, we explain the evaluation of the collected information in detail.

3.1. Evaluation of the collected information

In order to explain how we process data, we need to first explain our transfer policy. It is very similar to the transfer policy proposed by Dhakal et al. in [11]. It uses a threshold policy, which defines the excess workload of an agent based on its roles. This threshold policy is based on a queuing model, which characterizes the stochastic dynamics of the distributing of loads in a multiagent organization of n agents that 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. This plan is then 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_1, so the interarrival times are independent and identically distributed (i.i.d.) exponential variables with a mean of $1/\lambda_1$ [12]. We also consider that each agent performs requests according to a Poisson process of rate λ_2 and sends its workload information to the monitor agent at every sampling period.

During system initialization, a period is set for the organization. This period is called the sampling period, T, and is actually defined over the time window of $((k--1)T,\, kT)$, where k is the sampling instant. The workload of an agent at the kth sampling instant is as follows.

$$Q_i\,(k) = Q_i(k-1) - P_i(k) + R_i(k) - L_{ij}(k) \tag{1}$$

In Eq. (1), $Q_i(k)$ is the length of the ith agent's internal queue, where the plans are stored to be executed over a time window of $((k--1)T,\, kT)$. $Q_i(k-1)$ is the internal queue length of the ith agent from the previous sampling period. $P_i(k)$ is the number of plans performed by the ith agent over a time window of $((k--1)T,$ $kT)$. $R_i(k)$ is the number of requests received by the ith agent over a time window of $((k--1)T,\, kT)$. $L_{ij}(k)$is the excess load of the ith agent, transferred to the jth agent over a time window of $((k--1)T,\, kT)$.

Thus, each agent sends its workload data as a combination of the partial data, as mentioned above. If the multiagent organization is implemented by using a semantic web-enabled multiagent system framework (SEAGENT) [13], the components of Eq. (1) are described for the ith agent as follows.

$Q_i(k)$: The length of the ith agent's objective queue, where the goals extracted from incoming requests are stored to be scheduled over time window $((k--1)T,\, kT)$. $Q_i(k-1)$is the objective queue length from the previous sampling period.

$P_i(k)$: The number of executed plans of the ith agent over time window $((k--1)T,\, kT)$. We assume that the executed plan sends a response back to the clients.

$R_i(k)$: The number of newly arrived requests that are held in the incoming message queue over time window $((k--1)T,\, kT)$.

$L_{ij}(k)$: The excess load of the ith agent, to be transferred to the jth agent over time window $((k--1)T,$ $kT)$.

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 hierarchical task network (HTN) formalism [14,15], is shown in Figure 3. According to this plan, the workload data of each agent obtained from the workload message are transferred to the "compare current workloads" primitive task by an inheritance link. The monitor agent then computes the excess workload of each agent by comparing the agent's workload to the average overall load of the system. The excess workload of an agent at the kth sampling instant is as follows.

$$L_i(k) = Q_i(k) - \frac{\lambda_i}{\sum_{l=0}^{n} \lambda_l} \sum_{l=0}^{n} Q_l(k)$$

In Eq. (2), $L_i(k)$ is the excess load of the ith agent at the kth sampling period. $Q_i(k)$ is the workload data of the$i$th agent at the kth sampling period. λ_i gs the ith agent, which performs requests according to a Poisson process of rate λ_i. $Q_l(k)$ is the workload data of the lth agent at the kth sampling period. λ_l is the lth agent, which performs requests according to a Poisson process of rate λ_l.

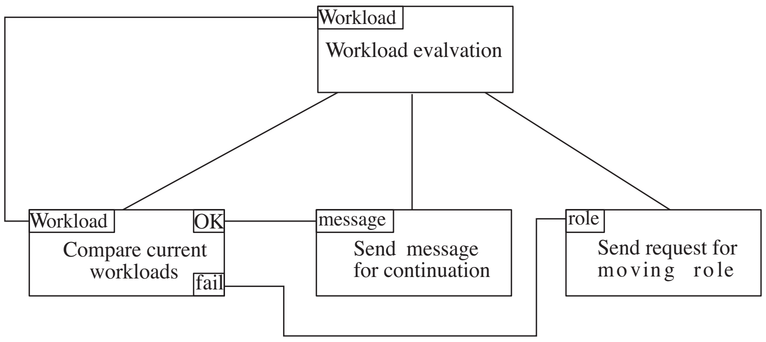
Workload
Workload evalvation
Workload
OK
Compare current workloads
fail
message
Send message for continuation
role
Send request for moving role

Figure 3. The "workload evaluation" plan in HTN formalism.

Eq. (2) is the fair share of the ith agent from the total workloads in the system. If the monitor agent detects that an agent has excess workload, then the monitor agent decides that the ith agent is a sender. According to the "workload evaluation" plan, the "compare current workloads" task returns the "fail" outcome with respect to the role that makes the agent overloaded. The role definition is admitted by the "send request for role moving" primitive task. In this task, the monitor agent prepares the message that includes the request for transferring the role that makes the agent overloaded and sends this message to itself to initiate the related plan. If there is no excess load condition for the agent, then the "compare current workloads" primitive task produces an "OK" outcome.

3.2. Moving roles

When the monitor agent receives the message that includes the request for role moving, it initiates the "role moving" plan shown in Figure 4. This plan takes the role that makes the agent(s) overloaded.

This plan of the monitor agent has the responsibility of moving the role that causes the agent to overload to one of the other agents in the multiagent platform. The plan takes the definition of the role that causes the agent's overloading and passes this provision to its subtask, named "determine ground agent." In the first subtask of this behavior, the agents that can play the role are identified according to the current state of the agents in the platform. As mentioned in the previous sections, the current state of every agent in the platform is stored in the platform ontology. The monitor agent can decide the proper agents for the role simply by checking this ontology during this task. In the second task, a request message is sent to the proper agents that have been selected in the previous task. This request message includes the role definition and asks a target agent if it is available to play this role. The "evaluate ground agent" task receives the response messages that are sent by other agents and selects one of the agents that have sent agreement messages corresponding to the request message as the ground agent. This task returns the agent identifier (AID) of the selected ground agent via the outcome named "OK."

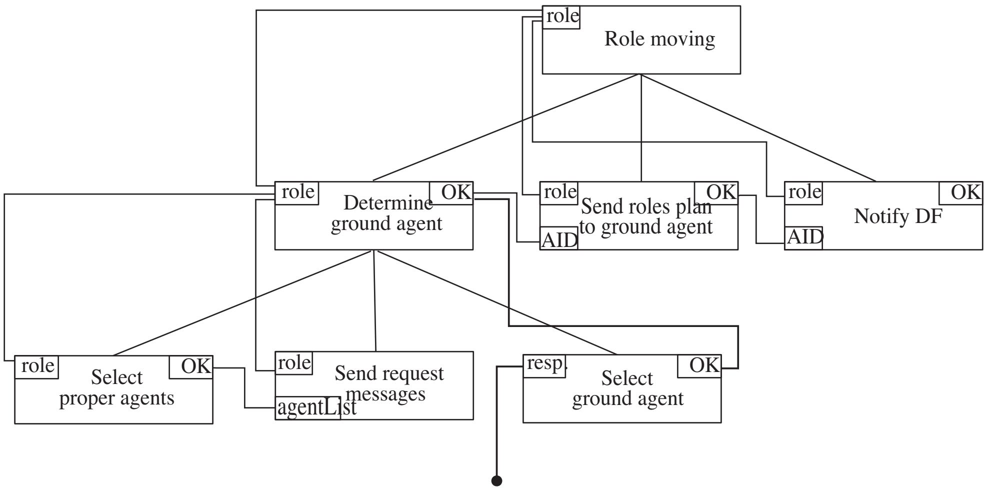

Figure 4. The "role moving" plan in HTN formalism.

The AID of the ground agent is taken in the other task, called "send role's plan to ground agent." This task has the responsibility of sending the plan library and knowledge base of the role to the ground agent.

The directory facilitator (DF) stores an ontology that includes the information about the roles that the agent plays in the platform. Hence, the changing of agent roles has to be communicated to the DF. The final task in the "role moving" plan takes the role definition and identification of the ground agent as provisions and sends a notification message to the DF.

At the end of the plan, the role that causes agent overloading has been moved to one of the other agents in the multiagent platform. Therefore, the load of the overloaded agent has been decreased by this refactoring operation. The agent with excess workload is notified to share its workload. When the overloaded agent receives the "notify" message, it will then send its excess workload to the new agent that plays this specific role.

4. Evaluation of the approach

The load-sharing approach presented in this paper was implemented with the SEAGENT multiagent development framework [16].

To show the effectiveness of our load-sharing approach, we applied it to an agent system application implemented by the SEAGENT research group. The case focuses on one of the core scenarios of the tourism domain. In this scenario, a traveler tries to organize a holiday plan that includes a hotel booking and transportation details. It is assumed that the 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's preferences. The primary roles that agents have in the scenario are identified as the "Traveler," "Travel Agency," "Hotel," and "Transportation Provider." In the initial design of the system, it was realized that the "Travel Agency" role is the most critical role in terms of the load-sharing perspective, because that agent has 2 critical responsibilities, hotel booking and selection of proper transportation. The agents with those roles in our design are shown in Figure 5.

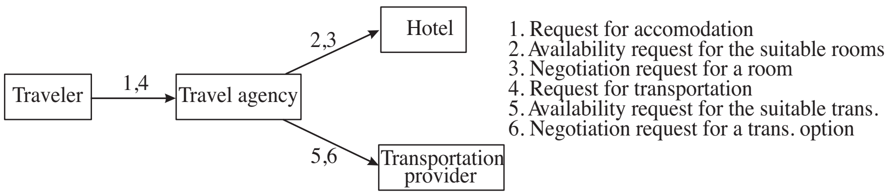
Traveler
1,4
Travel agency
2,3
Hotel
5,6
Transportation provider
1. Request for accomodation
2. Availability request for the suitable rooms
3. Negotiation request for a room
4. Request for transportation
5. Availability request for the suitable trans.
6. Negotiation request for a trans. option

Figure 5. Initial role diagram of the "prepare a holiday" scenario.

4.1. Discussion

For the evaluation of our approach, a test was performed on 2 computers with Intel Pentium 4 CPU running at 1.5 GHz and 512 MB of RAM, with Linux Ubuntu 6.10.

In this test, we evaluated the effectiveness of our load-sharing approach as the number of requests sent to the organization every 10 s increased. In the first case, while the system did not apply any load-sharing technique, we observed the response time of the agent that plays the "Travel Agency" role as the number of requests sent to it every 10 s increased.

In the second case, the agent that plays the role of the "Traveler" agent also sends its requests to the agent playing the "Travel Agency" role. The monitor agent receives workload messages sent by the "Travel Agency" agent and some other domain agents, and it evaluates the load of each agent based on their roles in the organization. When the monitor agent detects that an excess load for an agent exists, it finds a suitable agent that can play the "Travel Agency" role on another ground computer and sends the "Travel Agency" goal and plans to it. It then sends a transfer request to the "Travel Agency" agent. This agent transfers the excess load to the second "Travel Agency" agent by sending the received requests. In this test, while the system applied the load-sharing technique, we observed the response time of 2 "Travel Agency" agents as the number of requests each 10 s increased.

The results are illustrated in Figure 6. As can be seen, the response time of the agent increased without applying load sharing as the number of requests every 10 s increased. This result was expected, since the agent had to respond to more requests as the number of requests sent to it every 10 s increased. In the second test, the monitor agent detected that the agent became a sender at a point when the agent's workload exceeded the average load. It then decided to share the excess load with 2 agents that played the "Travel Agency" role.

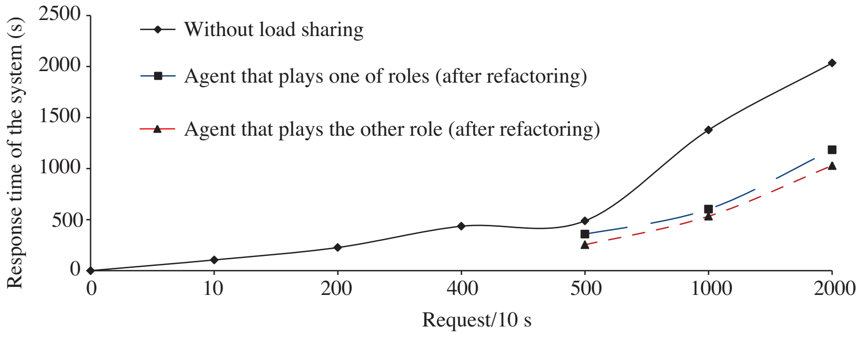

Figure 6. The effectiveness of the load-sharing approach.

Results of the second test are also shown in Figure 6. The system applying the load-sharing approach showed better performance compared to the system that did not apply load sharing, since 2 agents were running on different hosts and sharing the system's load to respond to sent requests.

5. Conclusion

In this paper, we presented a load-sharing approach for multiagent systems. We identified the agents that became senders and then decreased their workloads by distributing roles to more lightly loaded agents.

While evaluating this approach, we observed that the system applying load sharing outperformed the system without the load-sharing approach. In conclusion, the results indicate that efficient load sharing in multiagent organizations is sustainable using this model.

References

[1] K. Beck, Extreme Programming Explained: Embrace Change, Boston, Addison Wesley Longman Publishing, 1999.

[2] M. Fowler, Refactoring: Improving the Design of Existing Code, Boston, Addison Wesley, 2000.

[3] A. Schaerf, Y. Shoham, M. Tennenholtz, "Adaptive load balancing: a study in multi-agent learning", Journal of Artificial Intelligence Research, Vol. 2, pp. 475-500, 1995.

[4] S. Appleby, S. Steward, "Mobile software agents for control in telecommunications networks", BT Technology Journal, Vol. 18, pp. 68-70, 2000.

[5] J. Cao, D. Spooner, S. Jarvis, G. Nudd, "Grid load balancing using intelligent agents", Future Generation Computer Systems, Vol. 21, pp. 135-149, 2005.

[6] M. Singhal, N.G. Shivaratri, Advanced Concepts in Operating Systems, New York, McGraw-Hill, 1994.

[7] F. Giunchiglia, J. Mylopoulos, A. Perini, "The Tropos software development methodology: processes, models, and diagrams", Proceedings of First International Conference on Autonomous Agents and Multiagent Systems, pp. 35-36, 2002.

[8] F. Zambonelli, N.R. Jennings, M. Wooldridge, "Developing multiagent systems: the Gaia methodology", ACM Transactions on Software Engineering and Methodology, Vol. 12, pp. 317-370, 2003.

[9] FIPA, Review of FIPA Specifications, available at http://www.fipa.org/subgroups/ROFS-SG-docs/ROFS-Doc.pdf, 2006.

[10] Web-Ontology Working Group, Conclusions and Future Work, available at http://www.w3.org/2001/sw/WebOnt/, 2004.

[11] S. Dhakal, M.M. Hayat, J.E. Pezoa, C. Yang, D.A. Bader, "Dynamic load balancing in distributed systems in the presence of delays: a regeneration-theory approach", IEEE Transactions on Parallel & Distributed Systems, Vol. 18, pp. 485-497, 2007.

[12] A. Papoulis, Probability, Random Variables, and Stochastic Processes, 3rd ed., New York, McGraw-Hill, pp. 648-649, 1991.

[13] O. Dikenelli, R.C. Erdur, O. Gumus, “SEAGENT: A platform for developing semantic web based multi agent systems”, Fourth International Joint Conference on Autonomous Agents, pp. 1271-1272, 2005.

[14] M. Paolucci, O. Shehory, K. Sycara, D. Kalp, A. Pannu, “A planning component for RETSINA agents”, Lecture Notes in Computer Science, Vol. 1757/2000, pp. 147-161, 2000.

[15] M. Williamson, K. Decker, K. Sycara, “Unified information and control flow in hierarchical task networks”, Working Notes of the AAAI-96 Workshop “Theories of Action, Planning, and Control”, 1996.

[16] Ege University, SEAGENT: A Semantic Web Enabled Multi-Agent System Framework, available at http://www.seagent.ege.edu.tr, 2006.