AQTR 2004 (THETA 14)
2004 IEEE-TTTC - International Conference on Automation, Quality and Testing, Robotics
May 13 – 15, 2004, Cluj-Napoca, Romania

AN ABSTRACT LOAD-BALANCING ARCHITECTURE FOR LARGE SCALE MULTI-AGENT SYSTEMS

Lucian Buşoniu, Liviu Miclea, Enyedi Szilárd

Technical University of Cluj-Napoca
Department of Automation
26-28 Bariţiu Str., RO-400027, Cluj-Napoca, Romania
Tel.: +40-264-594469
E-mail: lucian_busoniu@yahoo.com, Liviu.Miclea@aut.utcluj.ro, Szilard.Enyedi@aut.utcluj.ro

Abstract: Multi-agent manufacturing systems and other large-scale scattered and distributed systems where multi-agent societies are being applied are often in need of an efficient task redistribution strategy in case of component faults or load variations of certain subsystems. The paper presents a simple, algorithmic approach for such a strategy, requiring low processing and communicational resources. Some specific concepts and the required elements of a multi-agent society that employs this strategy are introduced, followed by the abstract level presentation of the load balancing and related algorithms.

Keywords: multi-agent manufacturing, load balancing, intelligent agent, distributed system.

1. INTRODUCTION

Extremely dynamic customer requirements and global competition are shifting the production configuration of manufacturing organizations away from the traditional centralized, sequentially flowing planning, control and scheduling mechanisms. This approach is rendered too slow to adapt to evolving production styles and rapid variations in customer requirements, and limits the reconfiguration capability and the flexibility of the manufacturing system. The traditional, centralized organization may also easily lead to a large proportion of the system being shut down due to a single point of failure. The multi-agent society solution, being naturally distributed and decentralized, provides an easy method to overcome these disadvantages, and has been studied and employed in [4, 5] and [6].

However, reassigning tasks due to faults or load imbalances proves to be a difficult task even within such a society, and often results in wasted resources and poor system performance. This happens because the system architecture and the heuristics adopted by agents assigned to such production tasks are highly complex. Misbah and Fletcher [2, 3] have proposed a solution that attains load equilibrium in a multi-agent manufacturing system using concepts such as *temperature*, *relative heat* and *latent heat*. Employing the work accomplished in [1], we propose a solution to such a load-balancing problem, which makes use of a specific agent society structure and of simple computational algorithms to perform the load balancing.

Though primarily intended for flexible manufacturing systems, the viability of the approach extends to a wide spectrum of large-scale distributed, heterogeneous and/or geographically scattered systems, such as nationwide telecommunication or energy distribution networks. We present the concepts employed by the solution, the structure requirements an agent society must comply with so that the load balancing is applicable, and then detail the various necessary algorithms. Note that we shall not touch aspects such as implementation alternatives or inter-agent communication protocol specifications; we merely remain at an abstract, algorithm level for clarity.

2. CONCEPTS AND TERMS

First, it is assumed that the distributed system where the multi-agent society is applied is geographically partitioned – i.e., smaller, spatially separated and network-connected subsystems are working together. Second, the objectives of the distributed system can only be reached by the composition of a large number of tasks. Each of these tasks is performed by a particular type of agent, which requires certain skills in order to function correctly. All these skills together form a set. Some of the skills in this set are related, logically leading to a type of agent that exhibits them. We name this subset of related skills a *skill class* [2]. The bottom line is that an agent exhibiting a certain skill class may perform the duties of any other agent in that class. Such an agent is hereafter identified as a *task agent*, as opposed to the *service agents* employed by the solution.

A certain location L, corresponding to a specific geographical partition, requires a number of skill classes. A number of task agents of various types perform the system duties at that location. Let us consider now a single skill class at this location. This class contains a number of tasks:

$$C = \{t_1, t_2, \ldots, t_k\}. \tag{1}$$

A numerical score is assigned to each task, s_i, $i = 1..k$, directly proportional to the complexity of the task (which includes, but is not limited to, required resources and task execution time). Obviously, more tasks of a given type can exist at a certain location. We denote the number of tasks of type t_i with n_i, $i = 1..k$. If the number of task agents residing at the considered location and exhibiting skill class C is N, we define the *load factor* of the skill class C at location L as:

$$LF_{C-L} = \frac{\sum_{i=1}^{k} n_i \cdot s_i}{N}. \tag{2}$$

The purpose of the load-balancing now becomes clear: *to ensure a homogeneous load factor for every skill class over the locations requiring that class*. Of course, "homogeneous" is not an absolute term, and involves a tolerance interval. We assume that each task agent performs duties in a single skill class at a given time.

3. SOCIETY STRUCTURE

The required components of the multi-agent society are presented in figure 1. Locations are represented as thick-outlined rectangles, skill classes as ellipses and agents as labeled colored rectangles. We therefore have three locations, one of them requiring two skill classes, and the second and third requiring each only one skill class common to the first location. The *Directory Facilitator* (DF) agents implement the directory service specified in [7]. All agents register the services they offer with the local DF, which may also be queried by any agent for addresses of agents advertising a certain service. Therefore, a DF must be present at each network (geographical) location. DFs will federate over the network. An arbitrary number of *task agents* may be present at each location for the skill classes represented there.

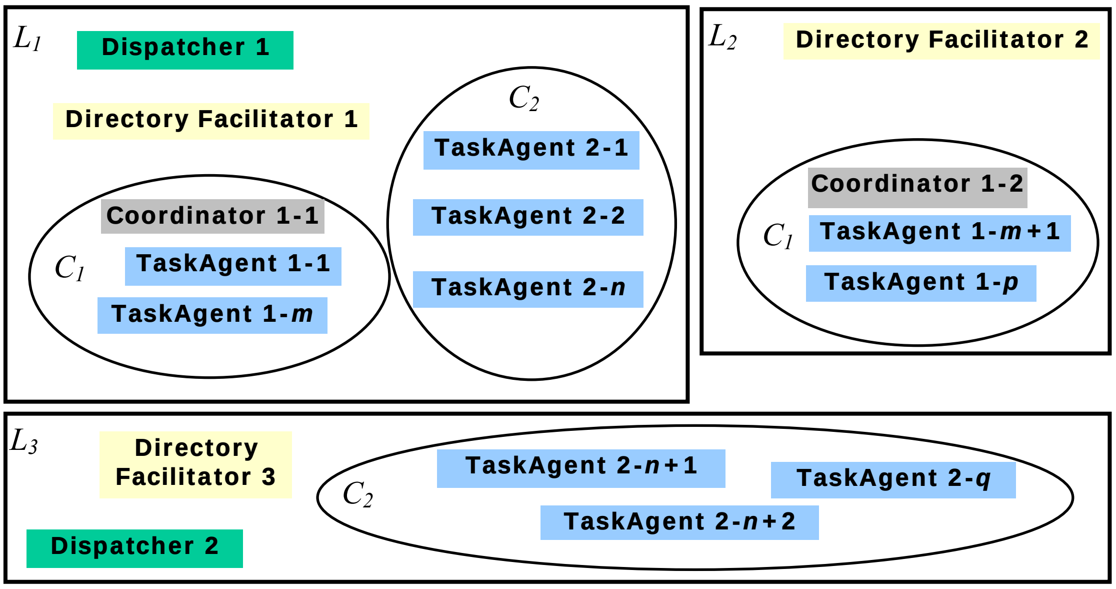

Figure 1. Required society structure

The key agent to our approach is the *Dispatcher*. The dispatcher performs the actual *load balancing*, using an algorithm we shall specify in the following sections. The purpose of the dispatchers is then to reach the following state of facts:

$$LF_{C1-L1} \cong LF_{C1-L2} \qquad LF_{C2-L1} \cong LF_{C2-L3} \tag{3}$$

There is no need for dispatchers to be present at each location. In fact, a single dispatcher would suffice, but in order to maintain the advantages of distributed processing and minimize the communicational overhead, more dispatchers may work at the same time within the society, assigned to sets of skill classes specific to a group of locations (e.g. parts storage warehouses). The task agents will then maintain contact with the dispatcher closest to their location.

Note also that C_1 makes use of a *Coordinator* agent, while C_2 does not. The coordinator is a generic concept representing the agent which performs tasks distribution within a certain skill class at a given location. If such an agent exists, it naturally knows the load factor of its skill class. The difference between the two types of classes is that the dispatcher will poll coordinators for load factors each time the load balancing algorithm is activated, but will need to be informed actively by self-coordinating testers of the tasks they are currently performing in order to maintain a load factor for those testers' class.

All task agents will need to register themselves with the DF as providing the skill class they are part of, so that the Dispatcher may find them when necessary. The coordinators also have to register as suppliers of the coordination service for their skill class. We do not specify names for these services, but leave this for the implementation.

4. SKILL CLASSES AWARENESS

The dispatchers make use of a skill classes maintaining algorithm which ensures the flexibility and scalability of the distributed system. Therefore they must become aware of new skill classes appearing within the system. The problem is in fact quite simple: for a skill class which does not employ coordinator agents, the dispatcher learns of its existence when the first tester performing the duties in that class begins functioning. The situation slightly changes if the class employs coordinator agents. The dispatcher will periodically poll the DFs about coordinator agents. The dispatcher becomes aware of a new class when finds a coordinator agent representing that class.

The problem of skill classes appearing / disappearing at subsystem locations is similar. Since the classes' registry is dynamic, there is no trouble changing it in order to reflect the system's state at the given time.

5. LOAD BALANCING ALGORITHM

The dispatcher will run a *load balancing thread* for each skill class under its supervision. This thread consists of a balancing sequence repeating at variable intervals. During every such sequence the following actions are performed by the dispatcher:

- if the skill class makes use of coordinators, they are polled about the load factor at their locations. Otherwise, the load factor at every location is already known from the information actively supplied by the testers.
- an array of load factors is formed, with a distribution of the values similar to what is represented in figure 2. The vertical bars symbolize the positions of the load factors.

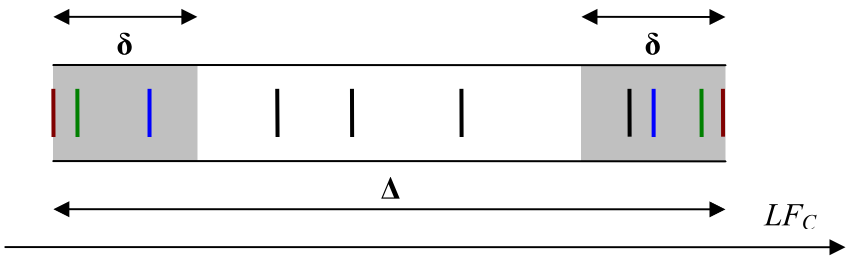

Figure 2. Load factors distribution

- the remainder of the sequence is executed only if the ratio of the greatest load factor to the least exceeds a *balancing activation threshold* specific to the skill class. Otherwise, the sequence is canceled.
- the value $\delta = d \cdot \Delta$, where *d* is a ratio coefficient chosen between 0.1 and 0.4, gives the intervals at the end of the load factors span where task agents transfers will be attempted. The dispatcher chooses a number of *pairs*, each pair formed by a location with the load factor at the lower end of the *Δ* interval, and one at its upper end. The pairs are distinctively colored in figure 2. The pairs forming stage ends either when one of the *δ* intervals has been exceeded or when a maximum number of pairs have been formed.
- for each pair, one task agent transfer is attempted from the less loaded location to the other. The agent transfer algorithm is detailed below.

The dispatcher will also ensure that the movement of the task agents does not become oscillatory, by never sending a task agent along a direction opposite to the direction a task agent of the same type was transferred at the last load balancing tick.

The dispatcher establishes the idle interval between two activations of a load balancing sequence (the *hyperperiod*) at every run, as follows: new load factors are estimated for the locations which exchanged task agents, and a new minimum and maximum are computed, together with their ratio *r*. The hyperperiod is then computed with the formula:

$$T = T_M - \mathrm{T} \cdot \left(1 - e^{\frac{r-\sigma}{\sigma}} \right), \; \mathrm{T} = T_M - T_m , \tag{3}$$

where T_m and T_M are, respectively, the minimum and maximum hyperperiod values, and σ is the balancing activation threshold. The effect of this formula is the rapid (exponential) approach of the hyperperiod toward its minimum as the load imbalance rises above the threshold. Obviously, when the balancing is not activated (the threshold is not exceeded), the hyperperiod is maintained at its maximum value.

6. TASK AGENTS TRANSFER

When the dispatcher desires to transfer a task agent exhibiting a certain skill class, it will query the DF for the presence of such agents at the less loaded location of the current pair (see the previous section). It will then randomly choose an agent and request it to transfer to the heavier loaded location.

A task agent is not required to comply unconditionally whenever a dispatcher requests it to move. If the agent knows that for some reason its remaining at the current location is critical, it will refuse the move. The dispatcher will mark such an agent in a registry it maintains and will not retry to transfer the agent for a certain amount of time, given in load balancing ticks.

If the agent is engaged in performing a task, it will respond with a “hold” message to the dispatcher, sending back an estimate of the time necessary to complete the task together with its availability to transfer upon completion. If this estimate does not exceed a ratio of the minimum hyperperiod, the dispatcher will agree and consider the agent transferred. The ratio value will be constant and chosen somewhere between 0.4-0.75. If, however, the completion time is not satisfactory, the dispatcher will choose another agent from the set returned by the DF and try to transfer it. This repeats iteratively until either an agent is scheduled for transfer, there are no more agents or a timeout occurs. In the latter two cases, the transfer attempt is canceled by the dispatcher.

7. CONCLUSIONS AND FUTURE WORK

A simple load balancing strategy for large-scale multiagent systems has been presented, together with the structure of the agent society it relies upon. The approach requires fewer resources in terms of communication bandwidth and processing time than the solution presented in [2, 3]. Provided that the various coefficients, ratios and timeouts influencing the algorithms are well-chosen, the stability of the strategy is guaranteed and the load factors for a skill class will converge into an interval situated around an average load factor after the various subsystem loads change or when faults occur. Therefore, in these stable states the system workload will be evenly distributed among task agents.

The first need to be addressed by future work is the implementation of a prototype and the study of its behavior under various simulation conditions. Without a doubt, the load balancing threshold and the ratio coefficient *d* have the strongest bearing over the efficiency of the strategy. In the current form, they are chosen for each skill class and remain constant in time. Alternatives of adaptation algorithms for these coefficients remain to be studied, together with their effects on the strategy performance.

REFERENCES

[1] L. Buşoniu (2003), *Multiagent Systems in DBIST and Distributed Testing*, eng. Diploma Thesis, Technical University of Cluj-Napoca, Department of Automation. Leader conf. dr. ing. Liviu Miclea.

[2] D.S. Misbah, M. Fletcher (2000), *Temperature Equilibrium in Multi-Agent Manufacturing Systems*, DEXA Workshop.

[3] M. Fletcher, D.S. Misbah (1999), *Task Rescheduling in Multi-Agent Manufacturing, Proceedings, Tenth International Workshop on Database and Expert Systems Applications*. DEXA, vol. 99, pp. 689-94.

[4] S. Handel, P. Levi (1994), *A Distributed Task Planning Method for Autonomous Agents in a FMS*, Proc. IEEE/RSJ/GI Int. Conf. on Intelligent Robots and Systems (IROS'94), Munchen, Sept. 12-16, 1994. Los Alamitos, CA: IEEE Computer Society Press, pp.1285-92, 1994.

[5] E.S. Tzafestas (1994), *Agentifying the Process: Task-Based or Robot-Based Decomposition?*, Proceedings 1994 IEEE International Conference on Systems, Man and Cybernetics (SMC'94), San Antonio, Texas USA, October 1994, pp. 582-87.

[6] W. Shen, N. Norrie (1998), *An Agent-Based Approach for Dynamic Manufacturing Scheduling*, Working Notes of the Agent-Based Manufacturing Workshop (1998), pp. 117-28.

[7] *** *FIPA SC00001, Abstract Architecture Specification,* http://www.fipa.org/specs/fipa00001/SC00001L.html.