Proceedings of the IASTED International Conference
ARTIFICIAL INTELLIGENCE AND APPLICATIONS
February 16-18, 2004, Innsbruck, Austria

An Intelligent Access Dispatching Mechanism Using Multiagent Framework

KAZUKO TAKAHASHI
School of Science&Technology
Kwansei Gakuin University
2-1, Gakuen, Sanda, 669-1337, JAPAN
email: ktaka@kwansei.ac.jp

CHIAKI KAWASHIMA*
Graduate School of Science
Kwansei Gakuin University
2-1, Gakuen, Sanda, 669-1337, JAPAN
email: chiaki_kawashima@hcc5.bai.ne.jp

ABSTRACT
In this paper, we describe the access dispatching mechanism in which access requirements from clients are regarded as autonomous agents. Each agent can observe only its local environment, and sends a message to get more information. Periodically, it computes the current best place, moves to the place, and sends the message telling the change of the environment to its neighborhood. The mechanism is robust against the change of environment, enables flexible access dispatching that increases performance and suppress computation and communication costs. We have implemented printers controlling system equipped with this mechanism, and performed simulation to show the effectiveness of the mechanism.

KEY WORDS
intelligent agents, cooperative AI systems, access dispatching

1 Introduction

Recent progress of mobile computing technology and enhancement of global communication infrastructure have brought about much attention to the mechanism such that multiple entities sharing distributed resources coordinate with each other to accomplish various jobs efficiently[1]. On realizing this mechanism we often face the problems of communication overhead and load balancing. One of the essential technologies to the efficient realization is access dispatching to the shared resources.

When multiple clients require to access a single resource, few problems may occur, since the requirements are put in order using a certain method. However, when alternative resources are available elsewhere, the problem is serious: if most of requirements are centered to a specific node, the performance of the total system would decrease.

For example, when you would like to use a network printer, if one printer is busy, it is more convenient to use another printer; when you would like to download an application software onto your computer, you may select a less-congested mirror site rather than the congested one to get the software fast.

Currently, lots of access dispatching mechanisms for load balancing have been developed and used [2][3][4]. Some of them are equipped with the intelligent control mechanisms: tiny task or urgent one is set to break in for immediate access, or tasks are removed from the site where fault has occurred.

However, once the requirement by a client is added to a queue of some node in a network, it never moves to another node, unless an unexpectable event such as a fault occurs. Therefore, when a big urgent task has broken in the queue of some node, it may take a long time for the predominant tasks to get their turns, despite that there exists another available resource. If there exists a server (or servers) which collects the requirements from clients, and which throws them one by one to the resource when it is ready, such a situation is avoidable. However, in such a case, the load of the server is so heavy, and less robust since it may break down.

In this paper, we propose an access dispatching mechanism in which access requirements from clients are regarded as autonomous agents. Our goal is to achieve high performance with low cost in the environment where multiple clients share multiple resources located at distributed nodes in a network, such as network printer and download site.

The mechanism is based on the multiagent framework[5][6]. Each agent only knows its immediate successor in the queue of a resource. When more information is needed, it sends other agents requirement messages to get it. Periodically, each agent selects the most appropriate destination at the instant, moves to the destination, and sends the message telling the change of the environment to its immediate successor. Receiving the update message, the agent updates its local information as well as sends the message telling the change of the environment with its own change to its immediate successor. In such a way, messages of updating are passed sequentially, and finally, all the agents have new information.

Using agents enables more flexible access dispatching and higher performance of the total system, and it ensures the robustness against the change of the environment. On the other hand, if large amount of communication is needed to grasp the change of the environment, communi-

*Currently, Nissay Information Technology Co.,Ltd.

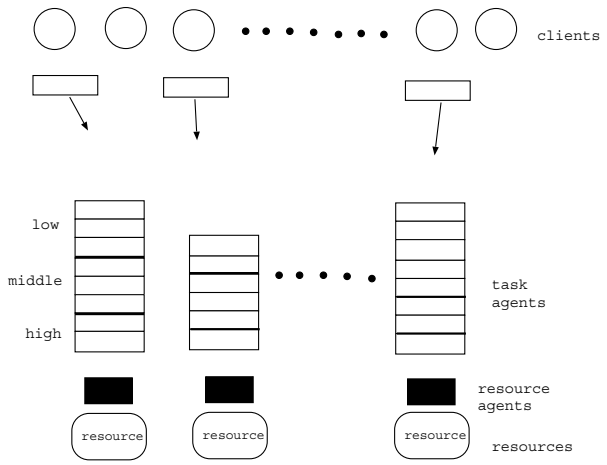

Figure 1. System configuration

cation overhead is too big to be ignored. In our mechanism, the message consists of task ID, waiting time and action, not including the whole state. Moreover, the number of occurring messages is small and the computation of agent is also simple. Therefore, the communication cost is low.

We show a printers controlling system equipped with this mechanism. We have implemented the system and performed a simulation. The result shows the effectiveness of the mechanism.

The paper is organized as follows. In section 2, we describe the idea and behavior of the access dispatching mechanism. In section 3, we show a printers controlling system with this mechanism and show the result of simulation. In section 4, we discuss the effectiveness of the mechanism. And in section 5, we show the conclusion and future works.

2 The Mechanism for Access Dispatching

2.1 The Idea

The mechanism which we propose here is based on the idea of multiagent with incomplete information.

The system consists of resource agents and task agents. These agents cannot see the whole state, but have only their local information. Figure 1 shows the configuration of the system.

Each resource has a queue in which task agents are waiting for an access to the resource. A resource agent stays at each resource watching the state of the resource, and when special events such as fault and restart occur, it recognizes them. It connects the resource and the top of the task agents in the queue by exchanging messages.

A queue of a resource consists of three parts depending on the priority. The part of high priority is processed first, the part of middle priority is second, and the part of low priority is the last. The tasks in the same part are processed in a first-in-first-out manner.

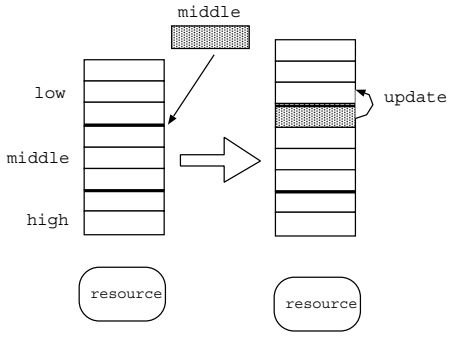

Figure 2. Creation of a task agent

A task agent is created at each time a client requires, and when the task is completed, it disappears. Each task agent can observe only its immediate successor in the queue. When needed, it asks other agents to give information, updates its local information, and decides its action to be taken next. A task agent has, as attributes, its priority, its own size and the ID of the physically/logically nearest resource from it.

2.2 The Procedure

Let n be the number of available resources located at some node.

When a client requires, a task agent A whose priority, size and the ID of the nearest resource are p, s and r, respectively, is newly created. It asks all the resource agents the current waiting time. Then, the resource agent asks the current waiting time to the task agent located in the last position in the part of the corresponding priority of the queue. Receiving the answer, the resource agent sends it to A. A decides the current best resource by considering waiting time, distance from the resource and the priority. Then, it moves to the last position in its priority part of the queue of the selected resource, updates its local information, and sends the message of requirement of updating the waiting time to its immediate successor $suc(A)$ (Figure 2). If there is no successor, $suc(A)$ is $none$.

In the followings, f denotes a function which computes the estimated time for processing the task of size s at node k.

the inference of task agent A at node S
(initially S is $none$)
T ← current_best_resource(A)
If T\=S
  move(A,S,T) // movement from S to T

decision of current best node for A

```
current_best_resource(A) {
For each k (1 ≤ k≤ n),
  B ← get_last(k,p)
    // let B be the last agent of priority p
  wait_time(k,A) ← wait_time(k,B) + f(k,A)
    // compute waiting time
  eval(k) ← comp_appr(wait_time(k,A),p,r)
    // evaluation of node k
T ← max of eval(k)
    // select the best node
return T
}
```

movement of A from S to T

```
move(A,S,T) {
If S\=none then
  remove_last(A,S,p)
    // update the local information of S
  ask suc(A) to do dec_time(f(S,A),A)
  C ← the agent that satisfies suc(C)=A
  ask C to do update_position(suc(A))
    // update the local information
    // of the previous position
If T\=none then
  B ← get_last(T,p)
  put_last(A,T,p)
    // update the queue of T
  suc(A) ← suc(B) // move
  wait_time(T,A) ← wait_time(T,B) + f(T,A)
    // update its own waiting time
  ask suc(B) to do add_time(wait_time(T,A),A)
    // update the local information
    // of the new position
  ask B to do update_position(A)
}
```

Periodically, the task agent re-computes the current best resource, and moves to it, if necessary. At the movement, it tells its departure to the current immediate successor, with the message of requirement of updating the waiting time. After the movement, it updates its own waiting time, and sends the message of requirement of updating the waiting time to the new immediate successor (Figure 3).

When a task agent which is not $none$ receives the message of requirement of updating the waiting time, it updates its own waiting time and sends the message of requirement of updating the waiting time with its own change to the immediate successor. The messages are passed sequentially to the task agent in the last position of the queue and the update of the waiting time for all the task agents in the queue has been accomplished.

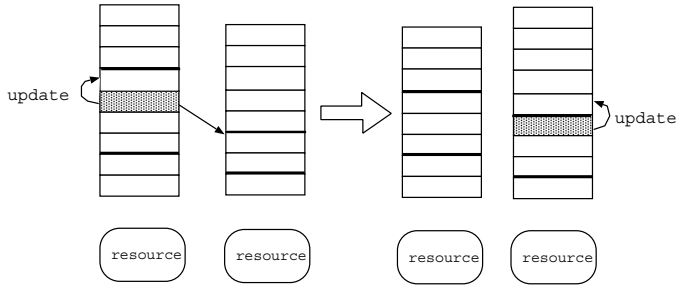

Figure 3. Movement of a task agent

task agent B receives the message
update_position(A)

```
(another agent comes as its immediate successor)
  suc(B) ← A
```

task agent B receives the message
add_time(wait_time(T,A),A)

```
(another agent breaks in)
wait_time(T,B) ← wait_time(T,A) + f(T,B)
  // update its own waiting time
ask sub(B) to do update(wait_time(T,B),B)
  // update the local information
```

task agent B receives the message
dec_time(f(T,A),A)

```
(the agent of its immediate predecessor has left)
wait_time(T,B) ← wait_time(T,B) − f(T,A)
  // update its own waiting time
ask suc(B) to do dec_time(f(T,B),B)
  // update the local information
```

A resource agent watches the task ID currently processed, and the state of the resource, namely, whether the normal processing is being done. When an access to the resource is broken because of fault, for example, the corresponding resource agent sets the waiting time to $unknown$ and tells it to the agent at the top of the queue. In addition, when an access becomes available again, it sends the message of resetting the waiting time to the agent at the top of the queue.

resource agent at fault

```
send update_time(unknown)
  to the task agent at the top of the queue
```

resource agent at restart

```
send add_time(0,resource)
  to the task agent at the top of the queue
```

When a task agent receives $unknown$, it recompute to decide the current best resource, and moves to it, if necessary. Agents whose size is large with the low priority

may remain. (Note that the break of access is assumed to be eventually released after some interval.)

task agent B receives the message
update_time(unknown)
wait_time(T,B) ← unknown
T ← current_best_resource(B)
If T\=S
move(A,S,T) //movement from S to T
ask suc(B) to do update_time(unknown)

When a task agent succeeds in accessing a resource and the task has been completed, then it tells its departure to the immediate successor, and after informing the client of its success, it disappears.

3 Printers Controlling System

We show a printers controlling system which has the mechanism described in the previous section.

Consider an office where several members are working, and they send printing requirements frequently. The members want to get their own printed documents from the nearest printer as fast as they can. When the nearest printer is busy, it is generally preferable to get the printed document fast even if it comes out from a far printer for a large urgent job, and to get it from the nearest printer for a small non-urgent job. Moreover, we hope all the printers may work with little idling time as possible as they can. Then, our goal is: the printers nearest from each client are used at a high rate, and the whole job is accomplished as fast as possible.

3.1 Simulation

We have implemented the printers controlling system using JAVA, and performed the simulation. A printer is regarded as a resource and a printing job is regarded as a task.

Figure 4 shows the screen-shot of the simulation. In this figure, the columns above each printer show the queue of task agents. Each block of the column is painted with different color depending on its priority. The height of a block corresponds to the size of each task. The figures below each printer show the remaining pages of the current task of printing. $wait$ denotes that there are no task agents in the queue.

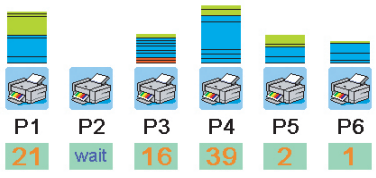

Figure 4. Screen-shot of simulation

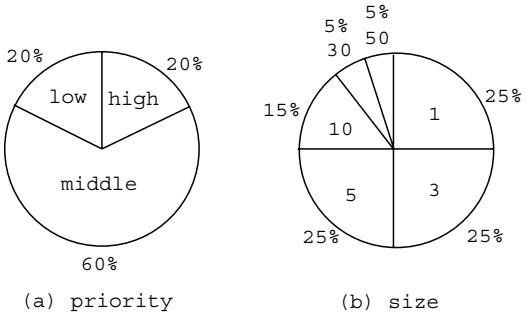

Figure 5. The rate of simulation data

	without fault	with fault
performance rate	111.7	128.3
achievement rate[high]	58.4	52.7
achievement rate[mdl]	77.9	67.0
achievement rate[low]	98.0	67.9

Table 1. Average number of performance rate(%) and achievement rate(%)

We perform the simulation under the following conditions.

1. Six printers and six clients are set.
2. All the printers are assumed to have the same performance (the same number of pages are printed out at a unit time).
3. The same number of requirements occur from all the clients at an arbitrary interval, and the total amount of the requirements is 100.
4. Size and priority of each task are set at random, according to the rate shown in Figure 5.
5. Every printer breaks down at random and after a certain interval, it restarts.
6. No delay caused by task movement is assumed.

We use two systems: (A) the controller with the new mechanism proposed in this paper and (B) a normal controller without re-computation nor re-movement. For these systems, simulation with given 30 data are performed for the case without fault and the case with fault, respectively.

The average values obtained as a result of 30 simulations are shown in Table 1. Performance rate denotes the rate t_A/t_B where t_A and t_B are the total amount time of printing for (A) and (B), respectively. Achievement rate denotes the rate of the tasks printed out from the nearest printers out of 30 data for (A).

3.2 Evaluation

As performance rate is above 100%, the proposed mechanism can decrease the total amount time of printing. About 17% difference appears comparing the case with and without faults. This means that the proposed mechanism is enough adaptive for the change of the environment. As for (B), tasks are loaded to a specific printer for some data. In such a case or in the case faults of printers frequently occur, the performance is much lower than that of (A).

In the case without fault, the rates of the task agents with high, middle and low priority that select the nearest printer are 40%,20% and 2%, respectively. This result reflects the setting that waiting time is taken to be more significant than distance, for a task with higher priority. Considering that the performance of system (A) is about 12% higher than that of system (B) even if far printer is selected, we should admit this setting.

In the case with fault, the achievement rates for (A) are low. It is because the task agents are forced to move to the other printer when a fault occurs. Instead, the performance is about 28% higher than that of (B).

Furthermore, although the average number is shown in Table 1, not a small difference between the results appears depending on each data. In the case without fault, the number of the task agents with high priority which select the nearest printer is big for some data, and small for other data. On the other hand, almost all of task agents with low priority select the nearest printer. In the case with fault, the numbers of the task agents which select the nearest printer are quite different depending on the frequency of fault occurrence.

If we choose the parameters of the function for evaluating the selected node, then we can obtain higher achievement rate. Moreover, a period of re-computation should be adjusted for the number of tasks and resources, and the average size of tasks, so that we can take advantage of this mechanism.

4 Discussion

4.1 The Characteristics of the Mechanism

In this mechanism, it is allowed that the task agents with a long waiting time move to other queue, and that the task agents with higher priority are broken in. It causes the waiting time of task agents to be changed, therefore the agents are forced to re-compute the current appropriateness of resources using new information. It enables more flexible access dispatching.

The outstanding characteristics of this mechanism is that it is not a server (resource agent) but a task agent that determines the destination. Although there are several studies on access dispatch in the distributed environment such as network printer or download server, no mechanism has been proposed in which access requirements are considered as autonomous agents. We use autonomous agents which are adapted to the change of environment. In addition, our mechanism is designed to suppress the computation and communication costs and to increase the performance of the whole system. An agent receives the message of task ID, waiting time and action, as an information on the change of the environment. Moreover, the number of occurring messages is small, the computation of an agent is also simple. Since an agent itself has an access requirement and the mechanism for computing the waiting time and deciding the best position, traffic of its movement is little enough to be ignored. Hence, low communication cost is realized.

4.2 Dispatching Accesses to Download Site

The mechanism can be applied to dispatching accesses to download site. When multiple clients require to download a certain application software, the requirements should be dispatched to the nearest (i.e.,short route, small number of hops) mirror sites. At that time, it is more preferable to take the estimated time for downloading and priority of the requirement into consideration. Generally, a server (or servers) takes intelligent dispatching control, but using agents can provide more flexible and more robust system.

4.3 The Possibility of Other Applications

We have studied the system in which multiagent with incomplete information cooperate with each other to solve a problem. Especially, we have interested in the case in that actions of some agents effect on the environments of the others. Such a type of problem is more difficult than a simple resource assignment problem[7][8][9], since an action of an agent may restrict on the next action of another agent. We have proposed the mechanism in which no manager process that can grasp the whole situation of the world exists, and agents exchange simple messages. It can suppress the computation and communication costs and increase the performance of the whole system. We have discussed the mechanism by taking a block loading problem as a case study [10]. In case of blocks, lower block agent cannot move nor receive another block agent on it when there is some block agent on it. On the other hand, as for the printers controlling system discussed in this paper, there is no restriction on moving of task agents from one queue to another queue. Both the systems have the same mechanism in the sense that autonomous agents which move from one place to another cooperate to solve the problem efficiently, but a block loading problem is an example with stronger restriction.

We can apply our mechanism to the treatment of mobile process with security by considering the security as a restriction of process movement. Moreover, it is sometimes more preferable to set the destination not as an absolute point of the node name, but as a relative point, such as, the same node with a specific resource. Our mechanism can

provide the assignment of relative point.

5 Concluding Remarks

In this paper, we have described the mechanism in which an access requirement is considered to be an autonomous agent when multiple clients share multiple resources.

The mechanism has the following advantages:

1. robustness against the change of environment
2. flexible access dispatching to increase performance
3. low computation and communication costs

We have implemented printers controlling system with the mechanism, and performed simulation to show its effectiveness.

In future, we are considering to apply the mechanism for controlling processes with security in distributed systems and controlling multiple robots.

Acknowledgements

This research is supported by KAKENHI14580434.

References

[1] Bellavis,P. and Magedanz,T. Middleware technologies: CORBA and mobile agents. *Coordination of Internet Agents*, Omicini,A, Zambonelli,F. Kluch,M. and Tolksdorf,R.(eds.) 110–152. Springer. 2001.

[2] CISCO: Local Director. http://www.cisco.com.

[3] IBM: Interactive Network Dispatcher. http://www.ibm.com.

[4] LINUX: Linux Virtual Server Project. http://www.linuxvirtualserver.org/.

[5] Weiss, G., ed. *Multiagent Systems*. The MIT Press. 1999.

[6] Wooldridge, M. *An Introduction to Multiagent Systems*. John Wiley and Sons, LTD. 2002.

[7] Decker, K. S. TAEMS: A framework for environment centered analysis & design of coordination mechanisms. *Foundations of Distributed Artificial Intelligence*, chapt 16. 429–448. 1996.

[8] Shehory,O. and Kraus,S. Task allocation via coalition formation among autonomous agents. *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence* Montreal, Canada, 1995. 655–661.

[9] Nair,R. and Tambe, M and Marsella,S. Role allocation and reallocation in multiagent teams: Towards a practical analysis. *Proceedings of The Second International Joint Conference on Autonomous Agents and Multiagent Systems*. Melbourne, Australia, 2003. 552–559.

[10] Kawashima, C., and Takahashi, K. A problem solving by multiagent in the environment with dense interaction. *Proceedings of 45th Annual Conference of Information Processing Society of Japan*, Tokyo, Japan, 2003. 301–302. volume 2. (In Japanese).