Distributed Market Broker Architecture for Resource Aggregation in Grid Computing Environments

Morihiko Tamai, Naoki Shibata[†], Keiichi Yasumoto, and Minoru Ito
Graduate School of Information Science, Nara Institute of Science and Technology
8916-5, Takayama, Ikoma, Nara 630-0192, Japan
{morihi-t,yasumoto,ito}@is.naist.jp
[†]Department of Information Processing and Management, Shiga University
Hikone, Shiga 522-8522, Japan
shibata@biwako.shiga-u.ac.jp

Abstract

In order to allow every user to extract aggregated computational power from idle PCs in the Internet, we propose a distributed architecture to achieve a market based resource sharing among users. The advantages of our proposed architecture are the following: (i) aggregated resources can be bought by one order, (ii) resource prices are decided based on market principles, and (iii) the load is balanced among multiple server nodes to make the architecture scalable w.r.t. the number of users. Through simulations, we have confirmed that the proposed method can mitigate the load at each server node to a great extent.

1 Introduction

The progress of high speed networks and high performance PCs in recent years has made it possible to realize high performance computing (HPC) such as Clusters and Grid computing at low cost. Over the past years, many projects like SETI@home [1] and GIMPS [2] are being conducted in order to achieve more than supercomputer power. These projects aggregate computation power from a large number of idle PCs connected to the Internet and use the aggregated power for large scale scientific computations. These systems can greatly reduce the management cost since each component of the system (i.e., PC) is maintained by the PC user. However, in order to make the system successful, an appropriate incentive for resource providers (PC users) such as high public interest, is required. On the other hand, NTT DATA and IBM began to provide a service called *cell computing*[3] where the required computation power is aggregated from PCs connected to the Internet and provided to business users at a cheaper cost than leasing a super computer. In this system, incentives for PC users are provided by giving them rewards such as electronic currency.

With the existing distributed computing systems, however, it is still difficult for each individual user to freely extract the required power for his/her personal objective. For example, a user may want to borrow relatively large computation power to encode a movie recorded with his/her camcoder into the MPEG-4 format. In such a case, it would be useful if a user could find idle PCs connected to the Internet and use their resources temporarily by paying something like virtual currency. The same user should be able to provide his/her resources to earn virtual currency while his/her PC is in the idle state at various times. Several ideas have been proposed to achieve fair resource sharing among users using the market mechanism [4, 5, 6, 7]. Here, the appropriate price of each resource is automatically decided depending on the balance between demand and supply.

When using the market mechanism, each user should be able to find the required amount of resources at as cheap a price as possible. Therefore, the existing approaches such as [4, 5, 6, 7] utilize a *market broker* (or *market server*) placed in a network node, which mediates between resource sellers and buyers. However, there are the following issues to make the market broker available for everyone. The first one is the scalability issue. When the number of users (orders) increases, the load of the broker also increases if it is implemented in a centralized server. So, we need a distributed implementation for the market broker. The second issue is the way of matching orders. The simplest way is to match one sell-order to each buy-order or vice versa. However, if a user wants to use relatively large computation power, it should be desirable to match a multiple set of sell-orders to one buy-order in order to increase availability of resources.

In this paper, we propose a distributed architecture for the market broker. The proposed architecture has the following advantages: (i) the load for processing orders is distributed among multiple server nodes, (ii) the architecture is fully distributed and needs no centralized control, and (iii) flexible matching such that one buy-order matches multiple sell-orders can be treated.

The basic idea of the proposed architecture consists of dividing the set of orders into multiple subsets based on the specified range of resources (e.g., the range of CPU power: 1000-2000MIPS) and allocating a server node to manage the orders falling into each range. Since each order may cover multiple ranges, if we let each server node find the pair of matching orders only in the subset, the result may differ from the case with a single centralized server. So, for each order covering multiple ranges, we replicate it and have the corresponding server nodes keep its replicas. We have developed a protocol for efficiently propagating replicas to the server nodes.

The rest of the paper is organized as follows. In Sect. 2, we briefly survey the existing techniques on market-based approaches for resource sharing. In Sect. 3, we explain about our framework for trading resources among users and how we integrate it into Grid computing systems. A distributed architecture for achieving the market broker is proposed in Sect. 4. We present the experimental results in Sect. 5. Finally, we conclude the paper in Sect. 6.

2 Related Work

Several techniques on market based resource sharing have been proposed so far.

In [8], a platform for mobile agents called D'Agents is proposed. Mobile agents execute their tasks on network nodes where they can move around. If the destination nodes belong to the differently administrated organizations, a control is required for deciding whether mobile agents can be accepted or not at those nodes. The decision can be made based on selling prices of resources at those nodes.

In [9, 10], resource sharing method for mobile ad-hoc networks is proposed. In a mobile ad-hoc network, since each node could be a router for receiving and forwarding other users' packets, incentives for router nodes are required. These methods provide incentive-based resource sharing techniques where packet forwarding services at router nodes are traded among service users at auctions.

The above existing methods focus mainly on achieving self-stabilization for restraining greedy users by keeping a balance of allocated resources among users. Our proposed method is different from these methods since our method allows users to trade any amount and any combinations of resources in PC Grid systems for their personal purposes using the market mechanism.

Some resource sharing methods have been proposed for PC Grid Systems such as Popcorn [4], Java Market[5] and JaWS[6]. In these methods, buyers of resources describe their applications in Java language, and register them to a node called the *market server*. On the other hand, sellers of resources execute these applications as Java applets by following URLs registered on the market server. The market server also conducts operations for uploading and downloading the application programs as well as for matching and retaining buy-orders and sell-orders. On the other hand, in [7], a scheduling system named Nimrod-G is proposed to solve a problem with time constraints in a given budget.

The above existing methods implement the market server as a simple client-server system. There are no detailed consideration to improve scalability of the market server or to assign user nodes for the market server autonomously. Our approach is different from the existing methods in the sense that we provide an autonomous and distributed architecture for achieving the market server.

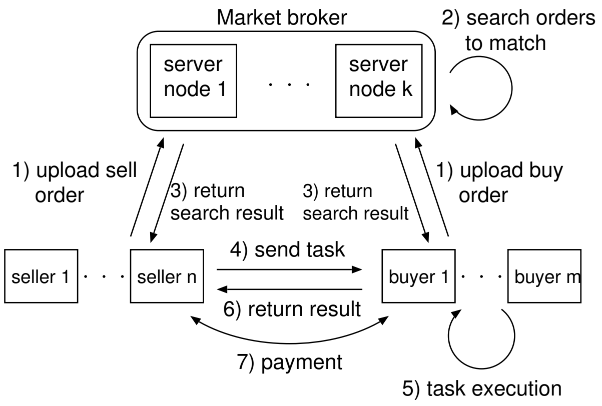

Figure 1. Framework for Market Based Resource Sharing

Distributed lookup services in P2P networks such as CAN[11], Chord[12] and Meghdoot[13] have been proposed so far. It may be considered to use these lookup services to search complementary orders on distributed market servers. On market servers, however, the best complementary order has to be found between a given value range. This kind of search is called *range search*. For example, if a buyer ordered a resource with more than 1000 MIPS, all of sell-orders with more than 1000 MIPS CPU power could be subjects of matching, and among them the lookup service has to find a sell-order with the lowest price. Since CAN and Chord do not support range search, it would be difficult to apply them to finding the best complementary order in a market based resource trading system.

On the other hand, Meghdoot supports range search. However, it does not support a selection mechanism, for example, to find a complementary sell-order with the lowest price. So, a client has to receive all of search results to find the best one. This may lead to poor scalability, since data size of search results increases as the number of sell orders retained in market servers increases.

Our proposed method is different from these existing methods in terms of the efficiency to find the best complementary order in distributed market servers, which is achieved by replicating the orders with the lowest (highest) price in multiple market servers in prior to other orders.

3 Framework for Market-based Resource Sharing

In this section, we describe outline of our resource sharing framework. We show the structure of our framework in Fig. 1. The framework consists of multiple sellers and buyers and a market broker consisting of multiple server nodes.

3.1 Components of our framework

Resource seller

Each seller receives a revenue by offering his/her computational resource to a buyer. Here, we treat only the following four types of resources: CPU power (in MIPS), memory amount (in MB), disk amount (in GB) and network bandwidth (in Kbps). Each seller represents his/her resource by a quadruple $r = (r_1, r_2, r_3, r_4)$, where r_1, r_2, r_3 and r_4 denote CPU power, memory amount, disk amount and network bandwidth, respectively. He/she also specifies the selling price $SellPrice$ of r, and the time period defined by beginning time $SellStart$ and ending time $SellEnd$, during which r is available. Each sell-order o^s can be represented by $o^s = (r, SellPrice, SellStart, SellEnd)$.

Resource buyer

Each buyer pays a fee for using computational resources offered by a seller (a set of sellers) to execute his/her task(s). Each task is represented as a combination of the following values: a set of computational resources (denoted by p) required to execute the task, the time duration (denoted by $ExecDuration$) for completing task execution, and the deadline (denoted by $Deadline$) of the task execution.

The set of required computational resources p is specified by a quadruple (p_1, p_2, p_3, p_4) similarly to the case of sellers. For example, a task with $p = (2000, 64, 0.3, 500)$, $ExecDuration$ = 3600s, and $Deadline$ = 9:00PM on Dec-01-2004 requires resources more than 2000MIPS of CPU power, 64MB of memory space, 0.3GB of disk space and 500kbps of network bandwidth, takes 3600 seconds to complete its execution and must be completed until 9:00PM on Dec-01-2004. He/she also specifies a budget $BuyPrice$ for the task and program/data code (or a pointer to the code) $Code$ of the task. Each buy-order o^b can be represented by $o^b = (p, BuyPrice, ExecDuration, Deadline, Code)$.

When a buyer wants to use a certain amount of resources from sellers, first he/she registers a buy-order o^b to the market broker.

Market broker

The market broker receives orders from multiple sellers and buyers, and finds the pair of complementary orders satisfying the conditions defined in Sect. 3.2. The matching of orders is conducted in the same way as the stock market at NYSE. That is, when the market broker receives a new order, it searches the best (e.g., the cheapest sell-order for a new buy-order) complementary order of all orders satisfying the conditions which are retained in its repository. If the broker finds the appropriate complementary order for the new order, it sends the matching result to the seller and buyer. If there are no complementary orders, the new order is retained in the repository of the broker.

3.2 Matching Condition

For each buy-order o^b and each sell-order o^s, if the following four conditions hold, we say that o^b and o^s can *match* to each other.

C1 $\forall i \in \{1, 2, 3, 4\}\ (o^b.p_i \le o^s.r_i)$

C2 $o^b.ExecDuration \le o^s.SellEnd - o^s.SellStart$

C3 $o^b.Deadline \le o^s.SellEnd$

C4 $o^s.SellPrice \le o^b.BuyPrice$

Let $\mathcal{S}(m)$ and $\mathcal{B}(m)$ denote the set of sell-orders and the set of buy-orders which a market broker m retains, respectively. We introduce the following two additional conditions to select the best order if multiple orders satisfy conditions C1-C4.

C5 (selection policy for buy-order) When m receives a new buy-order o^b and there are multiple sell-orders in $\mathcal{S}(m)$ which can match to o^b, one with the lowest sell price is selected.

C6 (selection policy for sell-order) Similarly, when m receives a new sell-order o^s and there are multiple buy-orders in $\mathcal{B}(m)$ which can match to o^s, one with the highest buy price is selected.

When the market broker receives a new order, it checks whether the order satisfies conditions C1-C4 or not. If there is no complementary order satisfying the conditions, the order is retained in $\mathcal{S}(m)$ or $\mathcal{B}(m)$. Otherwise, it selects one order based on condition C5 or C6.

One-to-many matching condition

When a buyer wants to execute a task requiring relatively large computation resources, it may be difficult to find a sell-order which satisfies conditions C1-C4 for the buy-order. If such a task can be divided into subtasks and those subtasks can be executed on different PCs, the availability of resources would be improved to a great extent. So, we extend our method to find the appropriate set of sell-orders whose sum of resources satisfies the conditions specified in a buy-order.

Let O^b denote a buy-order including n sub tasks (here, $n \ge 2$). We define O^b by $O^b.p = \{(o^b_1.p_1, o^b_1.p_2, ..., o^b_1.p_4), ..., (o^b_n.p_1, o^b_n.p_2, ..., o^b_n.p_4)\}$, $O^b.ExecDuration$, $O^b.Deadline$, $O^b.BuyPrice$, and $O^b.Code = \{code_1, ..., code_n\}$.

Here, we assume that there are n tasks where i-th task requires resources $(o^b_i.p_1, o^b_i.p_2, o^b_i.p_3, o^b_i.p_4)$ to execute program/data code $code_i$, all tasks have the execution durations and deadlines smaller than $O^b.ExecDuration$ and $O^b.Deadline$, respectively and a budget $O^b.BuyPrice$ is used to buy resources for all tasks.

For each buy-order O^b, if there exists a set of sell-orders $S = \{o^s_1, ..., o^s_n\}$ satisfying the following four conditions, we say that O^b and S can *match* to each other.

D1 $\forall i \in \{1, 2, ..., n\}, \forall j \in \{1, 2, ..., 4\}\ (o^b_i.p_j \le o^s_i.r_j)$

D2 $\forall i \in \{1, 2, ..., n\}\ O^b.ExecDuration \le o^s_i.SellEnd - o^s_i.SellStart$

D3 $\forall i \in \{1, 2, ..., n\}\ O^b.Deadline \le o^s_i.SellEnd$

D4 $O^b.BuyPrice \ge \sum_{i=1}^{n} o^s_i.SellPrice$

If there is no set of sell-orders satisfying conditions D1-D4 for O^b, the market broker retains O^b until such a set becomes available.

3.3 Task execution and payment

When the broker matched a new order to the appropriate complementary order, program code and data for the buyer's task are transferred to the seller's PC to be executed.

When transferring code, the following security problems may occur: (i) the buyer may send malware to the seller's PC, (ii) the seller may steal important information by observing task execution or may return forged computation results to the buyer. To prevent the above problem (i), we can introduce the sandbox mechanism used by java applets. In order to prevent the problem (ii), we can use voting or spot-checking techniques proposed in [14]. Also, some techniques in Trusted Computing [15] can be used to improve security.

When execution of task completes and results are sent to the buyer, the buyer must pay a fee to the seller. This can also safely be achieved with a tamper resistant execution environment as described in [9].

3.4 Outline of Our Distributed Architecture

As the number of orders from buyers and sellers increases, load of the market broker increases. So, we propose a distributed architecture for making the market broker scalable using the following policies:

(1) use multiple nodes (called *server nodes*, hereafter) to implement the market broker.

(2) choose server nodes from sellers autonomously.

(3) when some server nodes are overloaded, new server nodes are dynamically allocated and orders are distributed among these server nodes in order to increase scalability.

The above (2) can be achieved by letting users of the matched orders pay a commission fee. The fee is used to buy resources for managing the operations of the market broker from resource sellers. Therefore, every user has an incentive to be a server node.

The above (3) can be achieved by dynamically dividing the order space into multiple sub spaces and by assigning server nodes to sub spaces one-to-one so that the number of orders in each sub space is always less than a specified threshold.

In the following sections, we focus especially on the above (1). We solve the problem using replicas of orders and a protocol to efficiently propagate replicas among server nodes.

4 Distributed Market Broker Architecture

In this section, first we explain the algorithm for the case that one buying order matches one selling order. Then, we present extension for one-to-many matching in Sect. 4.6.

4.1 Representation of Orders

For each buy-order o^b and sell-order o^s, conditions C1-C3 in Sect. 3.2 can be represented as follows.

$$\forall i \in \{1, ..., 6\}\ (o^b.v_i \leq o^s.v_i) \tag{1}$$

where $\forall i \in \{1, ..., 4\}\ o^b.v_i = o^b.p_i,\ o^s.v_i = o^s.r_i$, $o^b.v_5 = o^b.ExecDuration, o^s.v_5 = o^s.SellEnd - o^s.SellStart$, $o^b.v_6 = o^b.Deadline$, and $o^s.v_6 = o^s.SellEnd$. In expression 1, the range of i is in general $\{1, ..., d\}$, where d is the number of resources. Hereafter, we denote $(o^b.v_1, ..., o^b.v_d)$ and $(o^s.v_1, ..., o^s.v_d)$ by $o^b.\vec{v}$ and $o^s.\vec{v}$, respectively.

For each value of i, we assume that the possible value range of $o^b.v_i$ and $o^s.v_i$ is decided in advance. We denote the range as $[min_i : max_i]$. Each order o can be represented as a point in a d dimensional space $R = [min_1 : max_1] \times \ldots \times [min_d : max_d]$, since $o.\vec{v} \in R$. We call R the *whole region*. Hereafter, we explain the case with $d = 2$.

In our algorithm, matching a new buy-order to an existing sell-order retained by the market broker is symmetric to matching a new sell-order to an existing buy-order. Below, we explain our algorithm using the former case.

4.2 Dividing Order Space into Subregions

Let M be the number of server nodes in the set of server nodes $\{m_1, m_2, ..., m_M\}$. In our method, we divide the whole region R into M subregions. One server node is assigned to each subregion. Let R_i be the subregion for which the server node m_i is responsible (Fig. 2).

When a user registers a new order o in the market broker, it sends o to one of server nodes, say, m_s. We suppose that each user knows at least one server node in advance. Then, m_s forwards o to an appropriate server node m_t such that $o.\vec{v} \in R_t$. This forwarding can be achieved with an existing distributed lookup service of CAN [11]. When each server node m_i received a new order o and o is not in the its subregion $[min_i, max_i]$, m_i finds the neighbor node m_j so that the central coordinate of sub-region $[min_j, max_j]$ is closest to $o.\vec{v}$ in terms of Euclid distance among all neighbor nodes, and forwards o to m_j. In order to implement the above lookup mechanism, we let each server node m_i keep regions and IP addresses of neighbor nodes as well as its own region in R. Transmission of an order from m_s to m_t requires $O(dM^{1/d})$ hops in average [11].

When server node m_t receives buy-order o^b such that $o^b.\vec{v} \in R_t$, it searches a sell-order which satisfies conditions C1-C4 in Sect. 3.2 for o^b. If there is no sell-orders satisfying the conditions for o^b, it appends o^b to the set of waiting buy-orders $\mathcal{B}(m_t)$ as described in Sect. 3. Similarly, when m_t receives sell-order o^s and there is no buy-orders satisfying the conditions for o^s, it appends o^s to the set of waiting sell-orders $\mathcal{S}(m_t)$.

With this approach, if orders are received by M server nodes uniformly, we can expect that the load of each server node is reduced to $1/M$ compared with the case with a single server node. However, if a buy-order o^b and a sell-order o^s are received by different server nodes, respectively, o^b and o^s cannot be matched. For example, in the case of $o^b.\vec{v} \in R_7$ and $o^s.\vec{v} \in R_6$, o^b and o^s are received by server nodes m_7 and m_6, respectively. Thus, o^b and o^s cannot be matched even though they satisfy conditions C1-C4. We cope with this issue by replicating some orders in multiple server nodes.

4.3 Order Replication

We explain our order replication algorithm using an example. First, let us suppose that only one sell-order o^s_1 is registered in a server node m_7. The *valid region* of a sell-order o^s is defined as $R(o^s) = [min_1 : o^s.v_1] \times [min_2 : o^s.v_2] \times ... \times [min_d : o^s.v_d]$. o^s_1's valid region is depicted as a rectangle in Fig. 3. In Fig. 3 to 6, each number in the parenthesis (e.g., o^s_1(40)) represents the selling price of the corresponding order.

Now, suppose that a new sell-order o^s_2 has arrived at the server node m_6. If o^s_2's valid region spreads over the set of server nodes $\{m_4, m_5, m_7, m_8, m_9\}$ as shown in Fig. 4. We make these server nodes hold replicas of the order o^s_2.

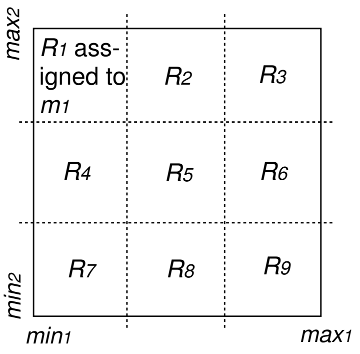

Figure 2. Server node allocation to each subregion

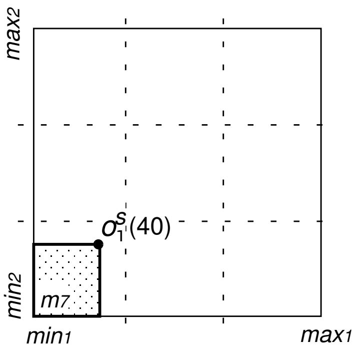

Figure 3. Sell-order o_1^s registered to server nodes

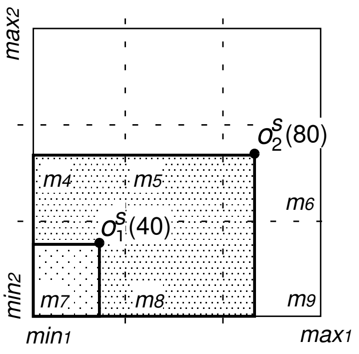

Figure 4. Sell-order o_1^s and o_2^s registered to server nodes

Next, we suppose that a new sell-order o_3^s has arrived at the server node m_2 and that o_3^s's valid region spreads over the set of server nodes $\{m_1, m_4, m_5, m_7, m_8\}$ as shown in Fig. 5.

In this case, however, server nodes m_7 and m_8 do not need to hold replicas of o_3^s. if a buy-order with price more than 80 is received by m_7 or m_8, the order first matches to order o_2^s since $o_2^s.SellPrice = 80$ is lower than $o_3^s.SellPrice = 90$. Thus, we make the set of server nodes $\{m_1, m_4, m_5\}$ hold replicas of the order o_3^s.

The details of our replication protocol is explained in Sect. 4.5.

4.4 Order Matching and Deletion

Let us suppose that a new buy-order o_1^b has been received by server node m_6 and that o_1^b has matched to o_2^s. Then, we remove o_2^s from m_6. Also, we remove order o_2^s's replicas from the set of server nodes $\{m_4, m_5, m_7, m_8, m_9\}$ as shown in Fig. 6. After deletion of the order o_2^s, order o_3^s becomes ready to match to buy-orders arriving at server nodes m_7 and m_8. So, we make these server nodes hold replicas of o_3^s.

Let us suppose that a new buy-order o_2^b has arrived at server node m_8 while the replicas of order o_2^s are being deleted after matching of o_2^s and o_1^b. In that case, o_2^b may match the replica of o_2^s in server node m_8, although o_2^s has already matched to o_1^b and been removed from m_6. In order to keep consistency, when a server node matches an order to a replica, we let the server node check if the original order exists or not. To do so, we attach the pointer to the original order in each replica.

4.5 Protocol for Order Replication and Deletion

In our protocol, we use messages $repl$ and del for replication and deletion of an order, respectively.

Replication message

When a server node m_i receives a new sell-order o^s, it calculates the *sub valid region* on each *downstream adjacent server nodes* of the order o^s. Here, sub valid region on m_i of o^s is defined as $R(o^s) \cap R_i$. And, the downstream adjacent server nodes of m_i are server nodes which own the neighbor regions of m_i towards the coordinate $(min_1, ..., min_d)$ in R. When the sub valid region of o_j^s and o_k^s *overlaps* on R_i and $o_k^s.SellPrice < o_j^s.SellPrice$ holds, overlapped sub valid region is *removed* from the sub valid region of o_j^s. That means o_j^s is not propagated to those overlapped sub regions any more (e.g., in Fig. 5, sub valid region of o_3^s is removed on m_7 and m_8 by o_2^s). The server node m_i checks whether the sub valid region of o^s is removed on each downstream adjacent server by the sell-orders retained by m_i, and if unremoved sub valid region found, m_i sends message $repl(o^s, m_i)$ to the downstream adjacent server node. Each replication message contains the content of o^s and the pointer to its original server node. When each server node receives replication messages for o^s, it checks whether the sub valid regions of o^s are removed on each downstream adjacent server node, and if unremoved sub valid region found, replication messages are forwarded to the downstream adjacent server nodes similarly.

For example, when a new sell-order o_2^s has arrived at server node m_6 in Fig. 7, replication messages for o_2^s are propagated as shown in Fig. 7. First, server node m_6 sends replication message $repl(o_2^s, m_6)$ to server nodes m_5 and m_9. Next, these servers forward the replication messages to server nodes m_4 and m_8. Finally, these server nodes forward the messages to server node m_7.

Deletion message

First, let us suppose that a new buy-order o^b satisfies the conditions with a replica of order o^s at server node m_i. As we explained in Sect. 4.4, before each server node matches an order to the replica, the server node checks if the original order exists or not. In this case, only if the original order of o^s exists, o^b can match to its replica at server node m_i, and consequently, the original and the replica of o^s are deleted.

Here, however, some replicas of o^s may be left at other server nodes. So, when original order o^s is deleted, we let its server node (say, m_j) send deletion messages $del(o^s, m_j)$ to its downstream adjacent server nodes similarly to the case of replication messages.

When each server node m_k receives a deletion message for replicas of an order o^s, it finds an sell-order $\hat{o^s}$ with the lowest price in the set $\mathcal{S}(m_k) - \{o^s\}$ and sends replication messages of $\hat{o^s}$ to downstream adjacent server nodes.

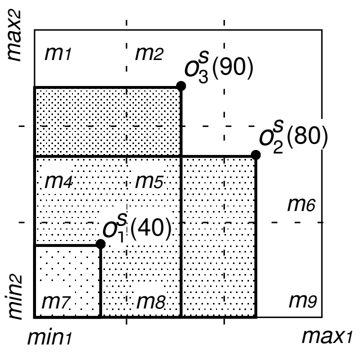

Figure 5. Sell-order o_1^s, o_2^s and o_3^s registered to server nodes

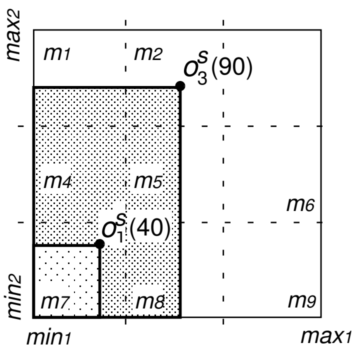

Figure 6. Sell-order o_1^s and o_3^s registered to server nodes

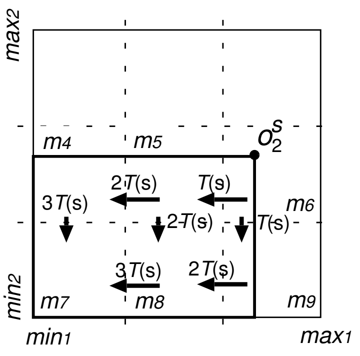

Figure 7. Propagation of the order o_2^s as time elapses

Introduction of time slot

In the above protocol, when a large number of orders are arrived or matched to complementary orders at one server node in a short time period, the server node may be overloaded due to operations for transmitting replication messages. In order to mitigate this problem, we introduce a constant time slot T to aggregate replication and deletion messages generated/received during every time interval T into a single message. Consequently, only one message is sent to each adjacent server per time slot.

When using a time slot, messages are transmitted as shown in Fig. 7. Each server node receives orders/replication messages during time slot T. In the next time slot, replication messages of the orders are sent to downstream adjacent servers.

However, using a large time slot may introduce large time lag of arrival times at server nodes far from the node which received the original order. That causes different results from the case using a single node for the market broker. However, we think that the difference is not so serious. In general, the selling/buying price increases as the selling/buying amount of resources increases. So, most of orders would match on the original server node or on server nodes near from the original.

4.6 One-to-many Order Matching

We suppose that there is n sub-tasks in a buy-order O^b. Let $\{o_1^b, ..., o_n^b\}$ denote the set of buy-orders for the sub-tasks. We call each o_i^b *sub-buy-order*. As we defined in Sect. 3.2, only the total budget is specified in O^b to buy resources for n sub-tasks. So, we register n sub-buy-orders in the market broker as independent orders, leaving their $BuyPrice$ fields *undefined*.

When a server node m_j such that $o^b.\vec{v} \in R_j$ receives a new order o_i^b such that $o_i^b.BuyPrice =$*undefined*, m_j finds a sell-order o_i^s with the lowest price which satisfies expression 1, and *tentatively* matches to o_i^b. Here, the tentative match means that the server node just retains the state of the matching until receiving the *confirmation* message without removing the order as a result of this matching. In this case, m_j sends the buyer the *tentative match* message with the value of $o_i^s.SellPrice$. If the buyer receives the *tentative match* messages from all server nodes where the sub-buy-orders were sent and $O^b.BuyPrice \geq \sum_{i=1}^{n} o_i^s.SellPrice$ holds, he/she sends the *confirmation* massage back to those server nodes. If the buyer's budget is less than the sum of the selling prices, the fact is shown to the buyer to prompt him/her to raise the budget.

When each server node receives the *confirmation* message for a tentative match which it retains, it checks whether the original order still exists or not, similarly to the one-to-one matching case. If the original order still exists, the order is confirmed and the *ack* message is sent to the buyer. Otherwise, the server node finds another sell-order which tentatively matches to the sub-buy-order, and sends a *tentative match* message to the buyer. Here, note that the order which has been confirmed cannot be tentatively matched to other orders.

If the buyer has received the *ack* messages from all the server nodes, O^b finally matches to the set of the sell-orders, and the buyer sends *commit* message to the server nodes. When each server node receives the *commit* message, it executes the similar operations to the one-to-one matching case. If the buyer receives the *tentative match* messages instead of *ack* messages from some server nodes, it tries to confirm them again.

If the buyer does not receive the *ack* messages from all the server nodes for a specified time period, it sends the *cancel* message to them to make the confirmed orders free. In this case, the buyer tries to register the order O^b from the scratch after a certain time period.

In the above algorithm, if multiple buyers register buy-orders with more than one sub-tasks and try to confirm their tentative matches simultaneously, there will be a conflict between confirmations for competing some common orders. For example, suppose that user1 and user2 registered buy-orders O_1^b and O_2^b and that they have tentatively matched to the sets of sell-orders $\{o_1^s, o_3^s, o_4^s\}$ and $\{o_2^s, o_3^s, o_4^s\}$, respectively. Then, suppose that user1 and user2 sent the *confirmation* messages at the same time and that the corresponding server nodes received the confirmation for o_3^s from user1 earlier than user2 and the confirmation o_4^s from user2 earlier than user1. In this case, neither user1 nor user2 can receive the *ack* messages from all the server nodes.

To avoid this problem, we adopted a policy that each buyer confirms tentative matches sequentially (i.e., it sends the confirmation message to the next server node after receiving the ack message from the previous one) in increasing order of server nodes' IP addresses.

5 Experimental Results

We conducted simulations on a PC in order to evaluate to what extent our distributed architecture can mitigate each server load. We measured the average number of messages processed by each server by changing the number of server nodes from 1 to 625.

The messages received and forwarded by server nodes consist of the following: (1) buy-orders and sell-orders sent from users, (2) replication messages for orders and (3) deletion messages for orders and replicas. In the simulation, 30000 orders (here, each order could be a buy-order or a sell-order at 1/2 probability) were sent to server nodes. The number of resource types was 2, and the amount to sell or buy each resource was given by uniform random values between 0 and 100. We used the same size for all server nodes' responsible sub-regions, and decided the message delay between server nodes at random between 10 to 160 ms. Selling price (buying price) was determined at random according to a Gaussian distribution whose mean and standard deviation are $(v_1 + v_2)$ and 30, respectively. Here, v_i denotes the amount of i-th resource to sell or buy.

We generated orders according to Poisson Arrival so that they arrive at server nodes every 10ms on average. The time slot was set to 1000ms. The experimental results are shown in Fig. 8. Fig. 8 also shows the average number of messages at each server node, and the maximum and minimum numbers of messages per one server node of all server nodes.

Fig. 8 shows that the number of messages processed by each server node can be reduced significantly by increasing the number of server nodes. We can see that the load is balanced among server nodes uniformly, since the deviation between the maximum and minimum numbers of messages and the average number is not so large in Fig. 8,

Next, we investigated the impact by introducing message aggregation in a time slot. We measured average number of replication and deletion messages received per second by each server node by changing length of a time slot from 0 to 10000ms. We used the same parameter values as the previous experiment. We show the result in Fig. 9.

Fig. 9 shows that we can reduce the number of control messages processed at each server node per second from 0.65 to 0.45 by increasing the time slot up to 10sec.

Next, we investigated the difference of matching results between the case with a single server and the case with multiple server nodes. In this experiment, we defined the user's satisfaction degree S as follows, and we compared the difference in the value of S.

According to conditions C5 and C6 in Sect. 3.2, when a buyer (a seller) sends an order to the market broker, it is desirable for the buyer (the seller) to match to a sell-order with lower price (a buy-order with higher price). Let us define $D = o_1.buyPrice - o_2.sellPrice$ if o_1 is a buy-order or $D = o_2.buyPrice - o_1.sellPrice$ if o_1 is a sell-order. If we use a single server node for the market broker, the complementary order which maximizes the value of D is always selected to match to o_1. On the other hand, if we use our distributed architecture, the value of D is not always maximized since there is a propagation delay of orders among server nodes. Let us define the satisfaction degree of each matching as $S = D_L/D_G$, where D_G is the maximum value of D when a single server node is used, and D_L is the value of D when our architecture is used. Here, $0 \le D_L \le D_G$. We suppose that $S = 1$ when $D_G = 0$. We measured average values of S for both cases, changing the number of server nodes between 1 and 625 and a time slot from 0 to 10000ms. We used the same parameter values as previous experiments. The result is shown in Fig. 10.

Fig. 10 shows that our architecture achieves almost the same satisfaction degrees (more than 0.95) as the single server node case for any number of server nodes. We see that the satisfaction degrees decrease when increasing the time slot interval. This is caused by the propagation delay of replicas among server nodes. We can improve the performance by propagating top-n replicas among server nodes, where n is the number of orders to be selected for the replication.

We also conducted the following experiment to evaluate efficiency of one-to-many order matching method described in Sect. 4.6. We measured probabilities in which all buy-orders are matched to sell-orders under the following two conditions. In the first condition, n buy-orders are registered to the market servers as sub-buy-orders using the method described in Sect. 4.6. In this case, the buyer's budget is the sum of all $BuyPrice$ of n buy-orders (buying price of each buy-order was determined by same settings as the above experiments) and matching fails if sum of all $SellPrice$ of complementary sell-orders exceeds buyer's budget or the buyer does not receive the *ack* messages from all the server nodes. In the second condition (simple method), n buy-orders (buying price was determined by the same settings of the first condition) are simply registered to the server nodes as independent orders. In this case, matching fails if there is a buy-order which is not matched complementary sell-order. We generated 15000 sell-orders, 13500 buy-orders which do not include sub-buy-orders, and 1500 buy-orders including sub-buy-orders. The probability distribution of the number of sub-buy-orders is exponential distribution whose average value is 10. Length of time slot was set to 1000ms. The number of server nodes is changed between 1 to 625. The results are shown on Fig. 11.

Fig. 11 shows that our method achieves higher matching probability than simple method regardless of the number of server nodes.

6 Conclusion

In this paper, we proposed a distributed architecture for a market broker which allows users to trade their resources based on the market mechanism. With the proposed architecture, each user can buy/sell a resource from/to the other user and also buy a larger amount of resource from multiple users by one order.

As we saw in Sect. 5, we confirmed that the proposed architecture is extremely scalable to the number of orders when using one-to-one matching for orders, and it greatly increases the availability of resources using the proposed one-to-many matching mechanism. Since we currently use

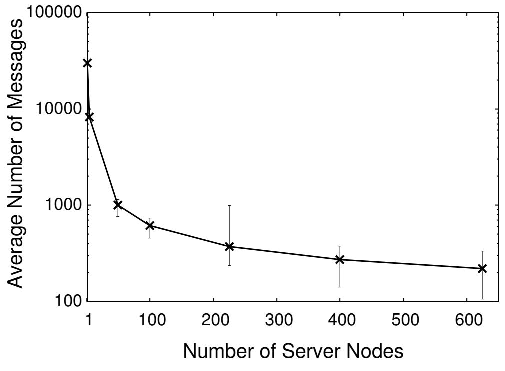

Figure 8. Average number of messages processed by each server node

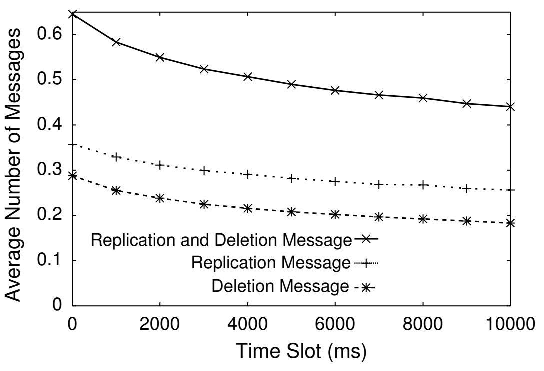

Figure 9. Time slot vs. average number of messages

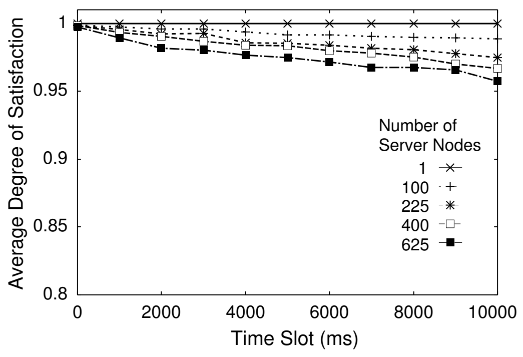

Figure 10. Average degree of satisfaction

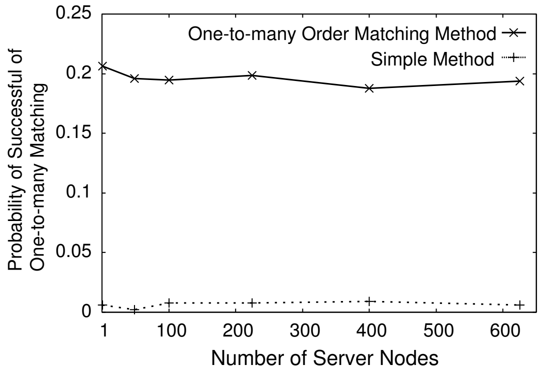

Figure 11. Improvement of availability

fixed ranges for subregions, a server node could be overloaded temporarily by receiving a lot of orders in a short time period. If such overloaded situation lasts for a while, the subregion should dynamically be divided to multiple regions recursively. Implementation of such a dynamic division mechanism is part of future work.

References

[1] SETI@home, http://setiathome.ssl.berkeley.edu/

[2] Mersenne Prime Search, http://www.mersenne.org/prime.htm

[3] cell computing, http://www.cellcomputing.jp/

[4] O. Regev and N. Nisan: The POPCORN Market - an Online Market for Computational Resources, Proc. of the First Int'l Conf. on Information and Computation Economies (ICE-98), pp. 148-157, 1998.

[5] Y. Amir, B. Awerbuch and R. S. Borgstrom: A Cost-Benefit Framework for Online Management of a Metacomputing System, Proc. of the First Int'l Conf. on Information and Computation Economies (ICE-98), pp. 140-147, 1998.

[6] S. Lalis and A. Karipidis: JaWS: An Open Market-Based Framework for Distributed Computing over the Internet, In Proc. of GRID 2000, pp. 36-46, 2000.

[7] D. Abramson, R. Buyya and J. Giddy: A computational economy for grid computing and its implementation in the Nimrod-G resource broker, Future Generation Computer Systems (FGCS) Journal, Vol. 18, No. 8, pp. 1061-1074, 2002.

[8] J. Bredin, D. Kotz and D. Rus: Market-based Resource Control for Mobile Agents, Proc. of the Second Int'l Conf. on Autonomous Agents, pp. 197-204, 1998.

[9] L. Buttyán and J. Hubaux: Stimulating Cooperation in Self-Organizing Mobile Ad Hoc Networks, ACM/Kluwer Mobile Networks and Applications (MONET), Vol. 8, No. 5, pp. 579-592, 2003.

[10] K. Chen and K. Nahrstedt: iPass: an Incentive Compatible Auction Scheme to Enable Packet Forwarding Service in MANET, Proc. of the 24th Int'l Conf. on Distributed Computing Systems (ICDCS 2004), pp. 534-542, 2004.

[11] S. Ratnasamy, P. Francis, M. Handley, R. Karp and S. Shenker: A Scalable Content-Addressable Network, Proc. of ACM SIGCOMM 2001, pp. 161-172, 2001.

[12] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek and H. Balakrishnan: Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications, Proc. of ACM SIGCOMM 2001, pp. 149-160, 2001.

[13] A. Gupta, O. D. Sahin, D. Agrawal and A. E. Abbadi: Meghdoot: Content-Based Publish/Subscribe over P2P Networks, Middleware 2004, 2004.

[14] L. F. G. Sarmenta: Sabotage-Tolerance Mechanisms for Volunteer Computing Systems, Proc. of the First ACM/IEEE Int'l Symposium on Cluster Computing and the Grid (CCGrid 2001), pp. 337-346, 2001.

[15] Trusted Computing Group, https://www.trustedcomputinggroup.org/home