An Autonomic Distributed Algorithm for Forming Balanced Binary Trees of Nodes in a Structured P2P System in a Multicast-enabled Environment

Takashi Yamanoue[†], Kentaro Oda[†] and Koichi Shimozono[†],

[†]Kagoshima University, Kagoshima city, Kagoshima, Japan

Summary
This paper describes an autonomic distributed algorithm which can be used to structure a group of nodes connected by TCP into a balanced binary tree, and an experimental structured P2P system which adopts this algorithm. This algorithm can be applied when nodes can be connected directly each other by TCP/IP, and IP multicast-able. When N nodes join group simultaneously, it takes O((log N)^2) time for all nodes to become members of the group, provided some conditions are satisfied. When a node in the group fails, the tree will be rebalanced by restarting the algorithm at the children of the failed node.
Key words:
peer to peer; overlay network; autonomic; distributed algorithm; binary tree.

1. Introduction

Managers of large Information and Communication Technology infrastructures (ICT infrastructures) frequently have to distribute identical software packages or data to a large number of terminals quickly. Reliability is important in such cases. For example, modern movie formats, such as MPEG2 and H.264, consist of key frames and subsequent changes between frames. If a terminal fails to receive a key frame during a real-time streaming movie, the image at the receiver terminal becomes corrupted for many frames. Thus, the reliability of the communication channel is important for today's digital movie broadcasting.
It has been shown that organizing the TCP connections between nodes on a switching network into a balanced binary tree is an effective way to quickly and reliably send large amounts of identical data[3].
We are developing SOLAR-CATS, a teaching tool for large computer laboratories[5][7][10]. This tool is equipped with a tool capable of quickly sending the image from one display in the class to all other displays. In order to implement this functionality, we organize the nodes receiving the data into a balanced binary tree where leaves are connected via TCP. This turns the system into a structured P2P System.

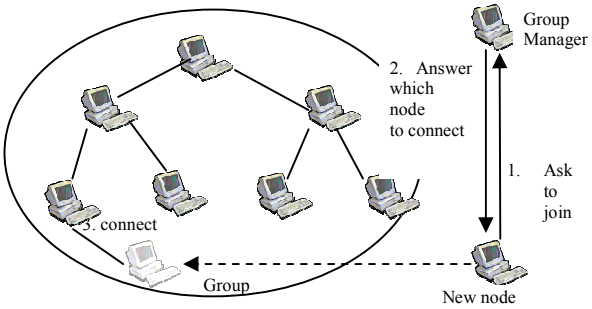
Manuscript received October 5, 2011
Manuscript revised October 20, 2011

Previous iterations of SOLAR-CATS used a group manager which instructs nodes joining the group to connect to an existing node in such a way that the binary tree structure stays balanced. However, the group manager represents a single point of failure. It receives all requests from new nodes. When a node in the group fails, children of the failed node also must query the group manager to determine which node they should connect to.
When a large number of students in a computer laboratory attempted to connect to previous versions of SOLAR-CATS simultaneously, the group manager would sometimes fail to form a group. Furthermore, the students had to enter the hostname or the IP address of the group manager into the GUI, making SOLAR-CATS more difficult to use.
In order to cope with such problems, we have created an algorithm that constructs a balanced binary tree from the nodes. We have also implemented this algorithm in SOLAR-CATS.

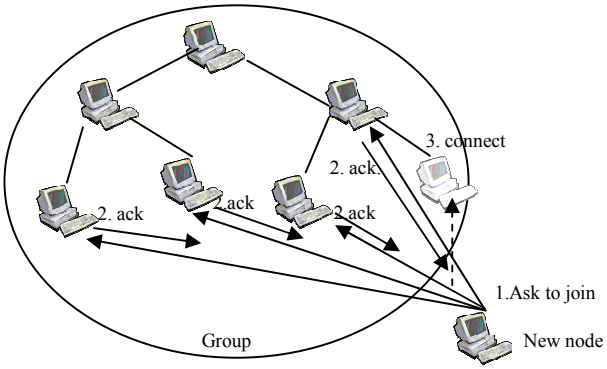
2. Algorithm

As mentioned in Section I, nodes in the group are organized into a binary tree and connected to each other using TCP. In the new algorithm, when a new node (requesting node) wants to join the group, the node broadcasts a request to join message (datagram) to all nodes in the group. Every *Waiting* nodes, who has less than 2 children in the group, send the requesting node an acknowledgement message. The requesting node then to connects to the waiting node that returns the acknowledgement first. If the node closest to the root node returns the acknowledge message faster than the other nodes in the group, the binary tree stays balanced. Figure 1 illustrates the previous algorithm and the new algorithm. Figure 2 describes the algorithm. Figure 3 and 4 are pseudo code for the procedures used by the new algorithm. In Figure 3, InitialRequestServer is a procedure that runs on all nodes in the group (waiting nodes). This procedure receives the request to join message and returns an acknowledgement message if the necessary conditions are satisfied. When the tree is first being constructed, this

procedure is running on the root node only. The procedure outlined in Figure 4, InitialRequestClient, is running on all requesting nodes. This procedure will stop after the requesting node connects to a waiting node as a child of that waiting node, and thus becomes a member of the group. After that, the node starts to run the InitialRequestServer procedure.



(a) Previous algorithm



(b) New algorithm

Figure 1. Previous algorithm and the new algorithm

```
begin
for each node co_begin
if this node is the root then
        InitialRequestServer;
else
begin
        InitialRequestClient;
        InitialRequestServer;
end.
co_end
wait until there is no node
which is running InitialRequestClient
end.
```

Figure 2. The Algorithm

```
Procedure InitialRequestServer
begin
    this_multicast_socket.join("mcast_port");
    repeat
      if  my_node.left_is_not_connected then
      begin
          message←
              this_multicast_socket.receive_a_request_message;
          wait_time( k*my_node.height);
      // In order to construct a balanced binary tree.
          the_remote_requester_address←
              message.source_address;
          this_tcp_socket.connect_to_(
              the_remote_requester_address , "recv_port");
          this_tcp_socket.send(
              my_node."address" ,  my_node."left_port");
          ack←this_tcp_socket.receive_ack;
          if ack== "accepted" then
              wait_until my_node.left_is_connected ;
      end;
      else
      if  my_node.right_is_not_connected then
      begin
          message←
              this_multicast_socket.receive_a_request_message;
          wait_time( k*my_node.height);
 // In order to construct balanced binary tree.
          the_remote_requester_address←
              message.source_address;
          this_tcp_socket.connect_to_(
              the_remote_requester_address , "recv_port");
          this_tcp_socket.send(
              my_node."address" ,  my_node."right_port");
          ack←this_tcp_socket.receive_ack;
          if ack== "accepted" then
              wait_until my_node.right_is_connected ;
      end;
      else wait_time(for_a_while)
    forever;
end
```

Figure 3. The Pseudocode for InitihalRequestServer

This algorithm terminates after all requesting nodes join the group. In other words, this algorithm terminates when there are no nodes in which InitialRequestClient is running. InitialRequestServer is running on all group member nodes. InitialRequestServer repeats the following steps in a busy-wait loop:

- If there is no child connected to the left side of the node, wait for a join request message from a requesting node. After receiving the message, return the IP address and port number for the left child to the requesting node using TCP after waiting for an amount of time proportionate to the height of the node in the binary tree of the group (in other words, proportionate to the distance from the root node). Then the node waits until it receives an accept message or a reject message from the requesting

node. If the node receives an accept message, the node waits until the requesting node connects to the left-side port.
- Similarly, if there is no child connected to the right-side node, the node executes the same steps as outlined above, but the requesting node connects to the right-side port instead of the left.
- If there are children on both the left and right sides, wait.

InitialRequestClient, which runs on requesting nodes, executes the following steps:
- Start the receiving thread to receive acknowledgement messages from the waiting nodes.

```
Procedure InitialRequestClient
begin
 (new receive_server_thread).start;
    this_multicast_socket.join(mcast_port);
    while my_node.up_node_is_not_connected
    begin
    this_multicast_socket.send(
           "request_to_join_in_the_group");
  wait_time( request_term);
    end;
end

thread  receive_server_thread
begin
 this_server_socket.
        start_receive_connection_at_port(recv_port);
 first_socket←this_server_socket.accept;
 (new accept_thread(first_socket)).start;
 while
  enough_term_to_receive_the_message_from_all_nodes
  begin
        rest_socket←this_server_socket.accept;
        (new reject_thread(rest_socket)).start;
  end

thread accept_thread(socket)
begin
   message←socket.read;
   socket.send_ack("accepted");
   my_node.connect_to_upper_node(
       message."address",  message."port");
   socket.close;
end
thread reject_thread(socket)
begin
   message←socket.read;
   socket.send_ack("rejected");
   socket.close;
end
```

Figure 4. The Pseudocode for InitialRequestClient

- Repeat the following until this node is connected to a waiting node.
  - Broadcast the join request message using IP multicast.
  - Wait for an enough time to receive the acknowledgement from an waiting node

The receiving thread executes the followings steps:
- Receive the first acknowledgement, return the accept message, and tell the requesting node to connect to the waiting node that returned the accepted acknowledgement.
- Wait until all acknowledgements are received, and return a reject message to all other waiting nodes that returned acknowledgement messages.

This algorithm realizes a distributed way of constructing a balanced binary tree. There is no single point of failure. When a node in the group fails, the group will reconfigure itself to keep the binary tree balanced if children of the failed node start the InitialRequestClient procedure after they stop their and their descendant's InitialRequestServer procedure. The balance will be subsequently improved by joining new nodes to the descendants of the failed node.

3. Proof and Time Complexity of the algorithm

We assume the following conditions for the algorithm.
- The statement
  message←
  this_multicast_socket.receive_a_request_message;
  which is executed when the node is waiting for child connections, always results in a message after the message was broadcasted p times. TCP connections and communications never fail between nodes, which is a reasonable assumption since TCP communication is quite reliable. This assumes that the use of network switch communication between two nodes over TCP does not affect any other TCP connections.
- InitialRequestServer and InitialRequestClient do not stop except at their normal termination.
- There are N nodes in the tree at time T.

A. Formation of a Balanced Binary Tree

When the number of nodes, N, is equal to 1, the only node in the group is the root node. This is a balanced binary tree. InitialRequestClient is not running at the root node and InitialRequestServer is running at the node. So the root node cannot be connected to another node. A requesting node can only be connected to a node in the group, the waiting node. A waiting node can have no children, one child, or two children. The graph of nodes and edges (connections) forms a binary tree. InitialRequestServer is not running on a requesting node. Therefore, the node cannot be connected to itself and it cannot be connected to another requesting node. This means that it is impossible

to create loops, and the requesting node can be only connected to a node in the group. As mentioned in Section II, a requesting node is connected to a node nearest to the root which has either no children or one child. The graph of nodes and edges (connections) in the group forms a balanced binary tree. The differences in distance between the root node and any two leaf nodes will be at most one.

B. Termination of the Algorithm

When the root node is the only node in the group, InitialRequestServer is running on the root node only and InitialRequestClient is running on all other nodes the algorithm will always terminate.

If the algorithm does not terminate, there will be a node on which InitialRequestClient runs forever. On this node, the procedure InitialRequestClient repeatedly broadcasts join request messages. Nodes in the group are always connected to form a binary tree as previously mentioned, and the procedure InitialRequestServer is always running on them. Therefore, there are always nodes which have no children (leaf node) or only one child node. These waiting nodes can receive join request messages. As we have assumed, waiting nodes always receive a join request message after the message was broadcasted p times if the nodes have no children or only one child. Acknowledgements are always returned to the requesting node. The requesting node always receives one acknowledgement and it will connect to the waiting node which returns the first acknowledgment. The InitialRequestClient will always stop after that because the loop terminates when the node is connected to the waiting node. The procedure can only continue to run the loop when there are no waiting nodes which have less than two children except when the procedure InitialRequestServer is not running on the root node. This contradicts our initial condition.

C. Time Complexity of the Algorithm

This subsection shows the theoretical time it takes to go from the time when only the root node is in the group to the time when all N nodes are in the group. We assume the following conditions in addition to the previous assumptions:

- When a join request message reaches a waiting node, the same message reaches all other waiting nodes simultaneously.
- When multiple requesting nodes nearly simultaneously broadcast join request messages, these messages reach all waiting nodes in the order in which they were broadcast.
- Communication time between two nodes is ignored.
- t_{ack} is the time between when a waiting node sends an acknowledgement and when the requesting node receives the acknowledgement and returns its accept or reject message.
- $t_{connect}$ is the time between when a requesting node receives an acknowledgement and when the node is connected to a waiting node.
- $t_{request\_term}$ is the interval between broadcasting a join request message and the next join request message.
- All other times are ignored.

A waiting node, which can be connected to a requesting node, waits to receive a join request message at the if statements for the left- and right-side connections. When a requesting node broadcasts a join request message and the available waiting node(s) at the closest available level from the root node receives the join request message, the waiting node returns the acknowledgement after waiting kh seconds where k is a parameter and h is the height of the waiting node. Then, the requesting node is connected to a waiting node. The waiting node may fail to receive some of join request messages. However, the node receives at least one message of the p messages which is broadcasted from the requesting node. So T_{h_1}, which shows the time between when a requesting node starts the procedure InitialRequestClient and when the node is connected to a waiting node, can be represented by the following inequality.

$$T_{h_1} \leq pt_{request\_term} + kh + t_{connect} \quad (1)$$

When the group consists of more than two nodes, and when two requesting nodes broadcast a join request message, T_{h_2}, which shows the time between when the two nodes start InitialRequestClient and when the two nodes are connected to nodes in the group, can be represented by the following inequality

$$T_{h_2} \leq pt_{request\_term} + kh + t_{ack} + pt_{request\_term} + kh + t_{connect} \quad (2)$$

In the worst case, when the second connection has to wait until the first connection finishes, we can use this as our upper bound. Two waiting nodes receive the first join request message from one requesting node simultaneously. So both nodes wait kh and return an acknowledgement. However, only one acknowledgement is accepted by the requesting node. The second waiting node can subsequently receive join request messages from other requesting nodes.

When there are m waiting nodes, whose heights are h and which can be connected by a requesting node, and 2m requesting nodes start executing InitialRequestClient simultaneously, T_{h_m}, which is the time between the start and when all 2m requesting nodes joined to the m nodes of the group at the height h, can be represented by the following inequality.

$$T_{h_m} \leq 2pmt_{request\_term} + 2mkh + (2m-1)t_{ack} + t_{connect} \quad (3)$$

In this inequality, $m \leq 2^{h-1}$. So

$$T_{h_m} \leq p2^h t_{request\_term} + 2^h kh + (2^h - 1)t_{ack} + t_{connect} \quad (4)$$

When h is the height of the balanced binary tree with N nodes, the total time of the connection T, which is the time from the start of the connection to the time when all N nodes are connected, satisfies the following inequality.

$$T \le \sum_{h=1}^{\lfloor \log N \rfloor} \{p2^h t_{request\_term} + 2^h kh + (2^h - 1)t_{ack}\} + t_{connect} \quad (5)$$

In this inequality,

$$N < 2^h, h \le \log N + 1 \quad (6)$$

$$\sum_{h=1}^{\lfloor \log N \rfloor} 2^{h-1} \le N - 1 \quad (7)$$

$$2^{\lfloor \log N \rfloor} \lfloor \log N \rfloor \le \sum_{h=1}^{\lfloor \log N \rfloor} h2^h < 2N(\log N + 1) \quad (8)$$

So,

$$T < p2(N-1)t_{reques\_term} + 2kN(\log N + 1) + N-1tack+tconnect \quad (9)$$

The time complexity T is $O(N \log N)$ which is larger than $O(N)$. However, if a multicast message does not reach all waiting nodes, much of this processing can be performed in parallel processing, resulting in a time complexity potentially better than $O(N)$.

4. Improvement of the Algorithm

As mentioned in the previous section, the algorithm can be improved. This section shows how we improved it.

In order to restrict the range in which multicast messages are propagated, a random number is added to the join request message of the InitialRequestClient procedure. A waiting node only receives messages where the (h-1) least significant bits of the message's random number matches the (h-1) least significant bits of its ID. The ID of the waiting node is equal to two times its parent's ID if the node is the left-side child. It is two times its parent's ID plus one if the node is the right side-child. The ID of the root node is one. If all waiting nodes receive the join request message in q times of its broadcasting from a requesting node and if (h-1) least significant bits of any two random numbers of them do not match the ID of one waiting node, 2^{h-1} waiting nodes can simultaneously accept join requests. T_h, which is the time between the start and when all 2^h requesting nodes joined to 2^{h-1} nodes of the group at the height h, can be represented by the following inequality.

$$T_h \le q\frac{p2^h t_{request\_term} + 2^h kh + (2^h - 1)t_{ack}}{2^{h-1}} + t_{connect} \quad (10)$$

$$T_h \le 2\{pqt_{request\_term} + qkh + q\left(1 - \frac{1}{2^h}\right)t_{ack}\} + t_{connect} \quad (11)$$

The total time T is shown by the following inequality.

$$T \le 2\sum_{h=1}^{\lfloor \log N \rfloor} q\left\{pt_{request\_term} + kh + \left(1 - \frac{1}{2^h}\right)t_{ack}\right\} + t_{connect} \quad (12)$$

There are following rules.

$$\sum_{h=1}^{\lfloor \log N \rfloor} h = \frac{\lfloor \log N \rfloor(\lfloor \log N \rfloor + 1)}{2} \quad (13)$$

$$\sum_{h=1}^{\lfloor \log N \rfloor} \frac{1}{2^h} \le 1 - \frac{1}{N} \quad (14)$$

So,

$$T \le 2p\{q\lfloor \log N \rfloor t_{reques\_term} + k\frac{\lfloor \log N \rfloor^2 + \lfloor \log N \rfloor}{2} + (\lfloor \log N \rfloor - 1 + \frac{1}{N})t_{ack}\} + t_{connect} \quad (15)$$

$$T \le pk\lfloor \log N \rfloor^2 + (2pqt_{request\_term} + pk + 2pt_{ack})\lfloor \log N \rfloor + 2p(\frac{1}{N} - 1)t_{ack} + t_{connect} \quad (16)$$

The above inequality shows that T has a time complexity of $O((\log N)^2)$. For sufficiently large values of N, this becomes less than O(N).

5. Experimental Implementation

We have implemented this algorithm in SOLAR-CATS. We are using this for a class with about 40 students and a seminar class with about 8 students. It is significantly easier to use compared to the previous version. Stability is also improved considerably. We have measured the time between when N-1 nodes start to join the group simultaneously after the root node establishes the group and the time when all N nodes become the members of the group. The values of N we used were 2, 3, 7, 15, and 31. Figure 5 shows our results. The horizontal axis represents the number of nodes using a logarithmic scale. The vertical axis represents the time using a square root scale, which is the time between the start and when all N nodes joined to the group, in seconds.

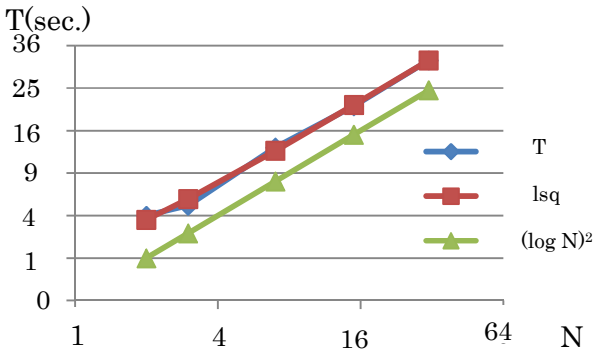


Figure 5. Time for making a group with N nodes

The curve labeled T shows the result time, T. The curve labeled *lsq* shows the quadratic curve fitted to the

results using the least squares method. The *T* and *lsq* curves almost overlap and are almost linear. This means that *T* is almost quadratic. The line which is labeled with $(\log N)^2$ is placed to illustrate how the curve of $\sqrt{T_{(sec.)}}$ is nearly linear in this graph.

It is clear that the curve of sqrt(T) is almost proportionate to log N. This means T is almost $O((\log N)^2)$.

We have taken k to be 1.0 second and $t_{request\_term}$ to be 0.5 second. The product kh is the time between when waiting node at height h receives a join message and when the node returns the acknowledgement for the message. $t_{request\_term}$ is the interval to repeat broadcasting the join request message at the requesting node. The fitted curve, *lsq*, can be calculated with the following equation.

$$T = 1.063(\log N)^2 + 0.839(\log N) + 1.719$$

This means p, which is the number times to receive a join request message by a waiting node, is almost one because k is 1.0.

We used a PC with the following specifications for each node in our experiments. All PCs were connected to a 100 Mbps network switch.

- CPU: Intel Core2 Duo E7300 2.7 GHz
- RAM: 2 GB
- OS: Windows XP Pro

SOLAR-CATS is written in Java. We used the Java SE Runtime Environment, version 1.6.0_21 to run SOLAR-CATS. In order to start the program automatically and simultaneously on all stations, we wrote a program using the PCs system clock, which is synchronized with a NTP server.

When we used the improved SOLAR-CATS in real classes, there were still a small number of nodes which could not join the group. We are currently investigating the cause of these problems.

6. Related Work

There is a wide body of research on reliable IP multicast which has produced such algorithms as TMTP (Tree-based Multicast Transport Protocol)[2], RMTP (Reliable Multicast Transport Protocol)[4] and the Reliable Multicast Tree construction algorithm[9]. These algorithms construct trees forming the communication channels between nodes. These algorithms, however, do not construct balanced binary trees.

BATON[6] is a structured P2P system which has a balanced binary tree structure. This is a distributed version of AVL tree. The paper on the BATON did not mention the time complexity for the case of many nodes joining the group simultaneously.

A P2P system proposed by one of the authors[8] is also a structured P2P system which has a balanced binary tree structure. This system does not need IP multicast. However, new nodes of this system have to know the address of the root node. When a node is going to join the group, it first contacts the root node before contacting any other nodes. So when many nodes are going to join the group simultaneously, the root node can potentially become a bottleneck.

7. Concluding Remarks

We have described an autonomic distributed algorithm, which can be used to make a group of nodes where nodes are connected by TCP to form a balanced binary tree. We have shown both theoretically and empirically that it takes $O((\log N)^2)$ time for all N nodes to become members of the group when N nodes join the group simultaneously. We then implemented this algorithm in a computer-assisted teaching system, SOLAR-CATS, and the system has been used in university classes. In real classes, there are still a small number of nodes which cannot take part in the group. We intend to to fix this problem in future work.

References

[1] Adelson-Velskii, G., E. M. Landis, "An algorithm for the organization of information". Proceedings of the USSR Academy of Sciences 146: 263–266. (Russian) English translation by Myron J. Ricci in Soviet Math. Doklady, 3:1259–1263, 1962.

[2] R. Yavatkar, et. al. “A Reliable Dissemination Protocol for Interactive Collaborative Applications”, Proc. ACM Multimedia, 1995.

[3] Takayuki Hirahara, Takashi Yamanoue, Hiroyuki Anzai and Itsujirou Arita, "SENDING AN IMAGE TO A LARGE NUMBER OF NODES IN SHORT TIME USING TCP", Proceedings of the ICME2000, IEEE International Conference on Multimedia and Expo, pp.987-990, New York City, USA, July 30-Aug.2 ,2000.

[4] Paul, S., Sabnani, K.K., Lin, J.C.-H., Bhattacharyya, S., “Reliable Multicast Transport Protocol (RMTP)”, IEEE Journal on Selected Areas in Communications, Vol. 15, Issue 3, pp.407-421 2002.

[5] Takashi Yamanoue, "A System which Shares the Common Operation on a Distributed System in Realtime Using P2P Technology", IPSJ JOURNAL, vol.46, No.2, pp.392-402, 2005.

[6] H.V. Jagadish, Beng Chin Ooi, Quang Hieu Vu, “BATON: A Balanced Tree Structure for Peer-to-Peer Networks”, Proceedings of the 31st VLDB conference, Trondheim, Norway, 2005.

[7] Yamanoue, T., “Sharing the Same Operation with a Large Number of Users Using P2P”, The 3rd International Conference on Information Technology and Applications (ICITA'05), IEEE CS Press, pp.85-88, July 2005.

[8] Takashi Yamanoue, Takeshi Nakamori, “A Structured P2P System Which Tries to Keep Its Balanced Binary Tree Shape by it-self”, IPSJ SIG Technical Reports, 2007-DSM-46, pp.55-60, Jul. 2007.

[9] Choonsung Rhee, Jungwook Song, Eujiun Kim, Sunyoung Han, “Reliable Multicast Tree Construction Algorithm”,

Mobility '08 Proceedings of the International Conference on Mobile Technology, Applications & Systems; 1-5, Ilan, Taiwan, 10-12 Sep. 2008.

[10] Takashi Yamanoue, "A Casual Teaching Tool for Large Size Computer Laboratories and Small Size Seminar Classes", Proceedings of the 37th annual ACM SIGUCCS conference on User services, pp.211-216, St.Louis, Missouri, US.. 11-14 Oct. 2009.

Dr. Takashi Yamanoue received his B.S. M.S. and Ph.D. in computer science from Kyushu Institute of Technology, Kitakyushu, Japan, in 1982, 1984 and 1993, respectively. He was a Ph.D. candidate of the Interdisciplinary Graduate School of Engineering Sciences, Kyushu University. He is a professor of the Computing and Communications Center, Kagoshima University. His research interests include P2P, distributed computing, compiler-compilers, web mining and computer assisted teaching systems. He is a member of IEEE, ACM, Information Processing Society of Japan(IPSJ), The Institute of Electronics, Information and Communication Engineers(IEICE), Japan Software Science Society(JSSST), the Robotics Society of Japan(JRSJ).

Dr. Kentaro Oda received the M.S. and Ph.D. from the Department of Artificial Intelligence, Kyushu Institute of Technology, Japan, in 1999, 2008 respectively. He is currently an assistant professor of Computing and Communications Center at Kagoshima University since 2009. His current research interests include adaptive middleware architecture, multi-agent systems (robotics soccer RoboCup), and distributed systems. He is a member of the ACM, IEEE (IEEE Computer Society).

Koichi Shimozono received the BE and ME degrees from Kyushu University, Japan, in 1991 and 1993, respectively. He is currently an associate professor in the Computing and Communications Center at Kagoshima University. His research interests include Japanese text processing, distributed systems, educational technology and internetworking.