

A Scalable Core Migration Protocol for Dynamic Multicast Tree^{*}

TING-YUAN WANG, LIH-CHYAU WUU^{**} AND SHING-TSAAN HUANG⁺

*Department of Computer Science
National Tsing Hua University
Hsinchu, 300 Taiwan*

*^{**}Department of Electrical Engineering
National Yunlin University of Science & Technology
Touliu, 640 Taiwan*

E-mail: wuulc@yuntech.edu.tw

*⁺Department of Computer Science
National Central University
Chungli, 320 Taiwan*

In past years, researchers have proposed the Core Based Tree (CBT) and Protocol Independent Multicast (PIM) protocols to route multicast data on the Internet. Such protocols need to locate the Core of a group to achieve efficient multicast routing. In this paper, we propose a scalable distributed protocol that can be used to move the Core to near-optimal location in the dynamic multicast tree, and that allows the Core to migrate efficiently when the multicast tree is expanded or shrunk. Our protocol does not require knowledge of the complete network topology, and information of all the members is distributed among local Agents; the Core only maintains the information of Agents of the group. Also, only the Agents participate in Core selection. Therefore, the proposed protocol reduces the runtime overhead and message complexity while performing Core migration.

Keywords: multicast tree, CBT, PIM, core migration, distributed algorithm

1. INTRODUCTION

Recently, group-based applications, such as Video Conferencing, Network Games and Distance Learning, become popular on the Internet. These applications require high data rates and considerably stringent delay constraints. Multicast [1] promises efficient use of network bandwidth for multiparty communication. In general, it is necessary to generate a multicast tree spanning all the group members in order to provide multicast service.

Some multicast routing protocols (e.g., CBT [2], PIM-SM [3]) use Group Shared Trees routing, where a group has an associated tree. A shared tree is built by choosing one node as the root of the tree; the root is called the *Core* [2]. The sender forwards data to the Core, and then the Core sends the data, which is then replicated as needed at each

Received May 15, 2002; accepted July 25, 2002.

Communicated by Biing-Feng Wang, Stephan Olariu and Gen-Huey Chen.

^{*} This work is supported by National Science Council under grant number 91-2213E-224-014. A preliminary version of the paper was presented at the 2002 International Conference on Parallel and Distributed Systems, Chungli, Taiwan.

branching point on the multicast tree reaching out to the group members. As indicated in [4], placement of the Core can influence the shape of the multicast tree and affect the performance of multicast routing. For example, if the Core is placed at the topology center of the group, delay variance can be minimized.

Most group-based applications allow members to join or leave dynamically. This results in the shape of a tree being changed, and the Core may no longer be located at the topology center. In this paper, a scalable Core migration protocol for a dynamic multicast network is proposed. The protocol not only places the Core as close to the topology center as possible, but also allows the Core to migrate efficiently when the multicast tree is expanded or shrunk.

An “optimal” core-based tree (OCBT) is chosen by calculating the actual cost of the tree rooted at each node in the network and picking the one that has the lowest tree cost and delay variance. We define the tree cost and delay variance in section 2. In practice, it is not feasible to construct all the possible trees for a given multicast group in a distributed environment. Therefore, most researchers try to find a near optimal Core. In the following, we present a brief overview of previous works.

Simple and administrative Core selection: Wall [5] proposed a *simple* method to construct a group-shared tree by first specifying a node near the topology center of the group. The group-shared tree is then constructed by merging the shortest path from the specific node to each member. In [6], Liu proposed an *administrative selection* method that uses a Core-Manager to keep track of the information of the candidate Core of each multicast delivery tree. The Core manager assigns one from a set of candidate Cores as a primary Core for a multicast group. This chosen Core is usually close to the majority of the group members. The Core manager and the set of candidate Cores are designated in advance.

There are two disadvantages with the simple method and the administrative selection method. First, while membership is dynamic, the methods cannot guarantee that the original Core is the best choice for the group. Secondly, these methods can locate the best Core easily when given complete topological information. However, in a distributed network like the Internet, topological information is often distributed across all nodes, such that no single node has complete topology information. Thus, the methods do not work if each node maintains local topology information and interacts with its neighboring nodes only.

Core migration in a distributed fashion: [7-10] separately proposed algorithms for migrating the Core in a distributed fashion. The methods they used probe the weights of neighbors and then pick a node with the minimal weight to continue probing. The weight function proposed is used to find an optimal Core based on various performance metrics. At each picking, the weight of the new probing node is always less than that of the old one. Finally, a better Core is chosen and migrated to the new location.

When a multicast group grows in a wide range network, the network becomes more complex, and the tree and the number of group members also become very large. Reducing the state space that the Core and each node should maintain is necessary for Core discovery in a distributed fashion to work. In [7-10], although complete topological in-

formation is not required, the probing node must have information of the multicast group members to compute the weights and find the new Core. This results in a heavy runtime overhead for computing the Core location and a large amount of space maintained by the Core.

This rest of this paper is organized as follows. In section 2, we give a brief description of the network model and the components with which we construct our Core Migration Protocol. In section 3, we describe how the Core Migration Protocol operates on dynamic multicast trees. Section 4 presents simulation results, and section 5 gives conclusions and future work.

2. THE PROPOSED APPROACH

In this paper, we propose a *Scalable Core Migration Protocol (SCMP)* for dynamic multicast trees. SCMP offers a distributed, flexible, and scalable approach to the construction of multicast trees that are shared among group members. Our approach has the following two merits:

- I. **It easily accommodates a wide range network in which group members are sparse and widely distributed on the Internet.** We partition the members into several subsets. Each subset forms a region that dynamically changes according to membership. In a multicast tree, a region is a subtree. The root of each subtree is chosen as an Agent to keep information of nodes in the subtree. One of the Agents is selected as the Core.
- II. **The state space that the Core requires is reduced, and the computing time for Core migration is less than it is with previous distributed Core discovering algorithms [7-10].** The Core keeps an Agent list rather than a member list. Compared with [10], the state space of the Core in SCMP is 85%~65% smaller. Only the Agents participate in Core selection. This decreases the running overhead and avoids control message congestion when the Core is migrated.

2.1 Network Model and Notations

The network is modeled as a simple, undirected, connected graph $N = (V, E)$, where V is a set of nodes and E is a set of edges. The nodes represent the *designated routers (DR)*, and the edges represent the links. The node is distributed randomly. Each node has information of only adjacent nodes. Each link l is symmetric with a nonnegative cost $C(l)$ and delay $D(l)$. For simplicity, we assume that $D(l) = C(l)$ for each $l \in E$. We also assume that no messages are lost in transit during normal operation.

The *actual cost* of a tree is defined as the sum of the costs of the links in the tree. Given a tree T , the actual cost of the tree T is computed as follows:

$$\text{Actual Cost}(T) = \sum_{e \in T} C(e).$$

The *delay variation* of a tree T is defined as follows:

$$\text{Delay Variation (T)} = \max_{u,v \in M} \left\{ \sum_{l \in P_T(u)} D(l) - \sum_{l \in P_T(v)} D(l) \right\}, \text{ where } M \text{ is the set of group members and } P_T(v) \text{ is the path from the tree root to node } v.$$

The following table shows the notations used in our SCMP protocol.

Notation	Meaning
<i>DR-x</i> :	a DR called <i>x</i> .
<i>Agent-x</i> :	an Agent called <i>x</i> .
<i>Core-x</i> :	a Core called <i>x</i> .
<i>MRT(x)</i> :	Multicast Routing Table of <i>DR-x</i> .
<i>MemT(x)</i> :	Member DRs table of <i>Agent-x</i> .
<i>AL(x)</i> :	Agent List of <i>Core-x</i> .
<i>Message_{x,y}</i> :	Message from source <i>x</i> to destination <i>y</i> .
<i>dist(x, y)</i> :	The path delay from <i>x</i> to <i>y</i> .
<i>PreHop(m)</i> :	The message <i>m</i> that comes from a router denoted by <i>PreHop</i> .

In SCMP, the Core is the root of the multicast tree, and an Agent is the root of a subtree. For any two nodes (*x, x'*) on the tree, *x* is an *upstream* node of *x'*, and *x'* is a *downstream* node of *x* if there is a path between *x* and *x'*, and *x* is closer to the Core than *x'*. If a DR has no downstream node, it is a *leaf* of the multicast tree. If all the downstream nodes of a DR are Agents, the DR is a leaf of some subtree. As shown in Fig. 1, Agents 1, 5, 7, and 9 are the roots of subtrees-0, 1, 2, and 3, respectively, and Agent 1 is the group's Core. DR 3, 6, 10, 12, 13, 15, 16, and 17 are leaves of the multicast tree and subtrees. DR 4 is not a leaf of the tree, but is a leaf of subtree 0. DR 2 is not a leaf of subtree 0 because one (DR 6) of its children is not an Agent.

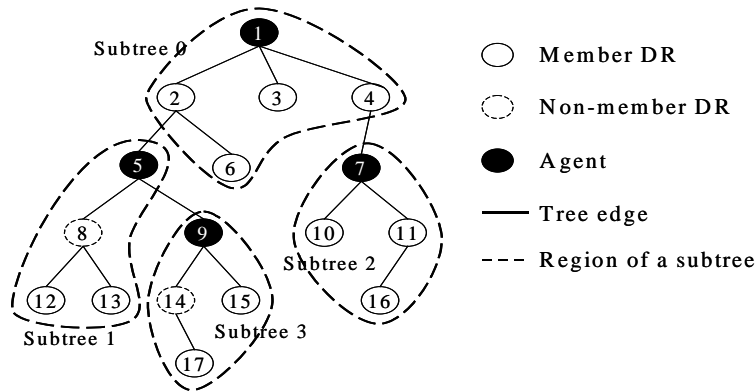


Fig. 1. A multicast tree topology example of group *G*. The delay of each link is 1.

2.2 Components

A. Designated Routers (DRs)

A protocol (e.g., IGMP) is assumed for routers to monitor the presence of group member hosts on their attached subnetworks and to propagate and exchange multicast information. For any multicast access LAN with two or more routers, there is a designated router (DR), just as in CBT [2] and PIM [3], that acts on behalf of the hosts on the LAN to start, join, or end a multicast service.

A DR that has member hosts attached to it is called a *member DR*; otherwise, it is called a *non-member DR*. To correctly deliver multicast packets, a DR on the tree must create an MRT entry for the group in its Multicast Routing Table (*MRT*) no matter whether it is a member DR or non-member DR. An MRT entry in a *MRT* has the following fields:

MRT entry = $\langle g, p, (c_1, c_2 \dots c_n) \rangle$, where g , which is an index of an *MRT* entry, is a multicast group address. In the multicast tree of group g , p is the parent of the DR, and child-list $(c_1, c_2 \dots c_n)$ contains the children of the DR. The DR forwards the packet to its upstream node through its parent or to its downstream node through its children when it receives a packet whose group address is g . For example in Fig. 1, DR 2's *MRT* entry is $\langle G, 1, (5, 6) \rangle$, and DR 17's *MRT* entry is $\langle G, 14, (\text{null}) \rangle$.

No DR needs to know which node is its Agent in the group. Logically, for any DR, its Agent is the first Agent on the path from the DR to the Core.

B. Agents

In SCMP, we choose the root of a subtree as an Agent. An Agent maintains a Table (*MemT*) for the member DRs in a subtree rooted at it, so that the Agent knows the entire member DRs in its subtree. *MemT* = $\langle (r_1, d_1), (r_2, d_2) \dots (r_n, d_n) \rangle$, where r_i denotes the DR i in the subtree, and d_i is the distance from the Agent to DR i . For the example shown in Fig. 1, the *MemT* of Agent 1 is $\langle (1, 0), (2, 1), (3, 1), (4, 1), (6, 2) \rangle$.

For any two Agents (x, x') on the tree, x and x' are *adjacent* if there is a path between x and x' , and if there is no other Agent on this path. For the example shown in Fig. 1, Agent 5's upstream adjacent Agent is Agent 1, and its adjacent downstream Agent is Agent 9.

C. Core

There is only one Core for a multicast group, and it is the root of the global multicast tree. One of the Agents is chosen as the Core. Hence, the Core is also an Agent, and it should do everything that an Agent does. In addition, the Core maintains a list of all the Agents of the group. The list is called the Agent list (*AL*), *AL* = $\langle a_1, a_2 \dots a_n \rangle$, where a_i is an Agent of the group. For the example shown in Fig. 1, the AL of the Core (Agent 1) is $\langle 1, 5, 7, 9 \rangle$.

2.3 Overview of SCMP

A. Objectives

The goal of SCMP is to place the Core as near as possible to the topology center and to migrate the Core efficiently when the multicast tree is expanded or shrunk. There are three primary objectives:

1. *Minimize delay variation*: Delay variation in the multicast tree will influence the routing performance [11]. The minimum delay variation provides synchronization among the various receivers and insures that no receiver is “left behind,” and that none is “far ahead” during the lifetime of the multicast session. A fixed Core cannot accommodate dynamic membership. This results in increased delay variation. It is unfair for group members to be subjected to large delay variation. Therefore, an ideal Core dynamically changes its location according to the present membership to minimize delay variation.
2. *Scalable*: In most Core selecting methods [7, 10, 12-14], the Core must maintain information of the group members to compute a new location. If the number of group members becomes very large, the scalability problem will arise. In this paper, the state space of the Core is reduced to make the SCMP scalable.
3. *Reduce overhead*: In [7-10], in order to find a new Core, each probing node and the neighboring node should compute their weights by querying all of the members/sources. This process leads to a heavy runtime overhead and message overhead. The SCMP reduces the number of querying actions and distributes the computing overhead to the Agents.

B. Tasks

The basic idea of our protocol is as follows: At the beginning of a multicast group, there is only one DR, one Agent, one subtree, and one global multicast tree. Of course, the DR is the Agent, and the Agent is also the Core of the group. Every DR can join or leave the group dynamically, and the Agent monitors its *subtree depth*, which is the distance between it and its farthest leaf. An Agent is created or removed depending on whether its subtree is expanded or shrunk. When the group grows, the Core may no longer be as the best location. The present Core chooses one of the Agents as a new Core if it is not at the best location. Our SCMP performs the following tasks, which are described in detail in section 3.

1. **Dynamic Membership**: The task is to add a node (DR) to a multicast tree or remove a node from a multicast tree. A DR can become part of the multicast tree by executing the Joining Process. A leaf on the tree can remove itself from the multicast tree by executing the Leaving Process.
2. **Agents Expanding and Shrinking**: We assume that each Agent continuously monitors its subtree depth. Once its depth exceeds the maximum delay bound, the Agent creates at least one Agent in its subtree. If the maximum distance from an Agent to a leaf of its downstream subtree is lower than the maximum delay bound, the subtree will merge with its downstream subtree, and the redundant Agent of its downstream subtree will be removed.
3. **Core Migrating**: Periodically, the present Core invokes the Core Migrating Process. The Core probes the weights of adjacent Agents and then picks an Agent with minimal weight to continue probing. Finally, a better location is chosen, and the Core is migrated to the new location.

3. SCALABLE CORE MIGRATION PROTOCOL

3.1 Dynamic Membership

We have designed the joining and leaving processes in a distributed fashion. When a group is created, the first DR of the group is assigned as the Core. When the last DR leaves a group, the group is terminated. In SCMP, all the DRs in the network can get group-to-Core mapping information from a *Group Name Server (GNS)*. The GNS is responsible for advertising itself to all the DRs in the network, maintaining an up-to-date (group, Core) associated list, and replying to Core-Inquirers with the designated Cores. When a group is created and terminated, or when a Core of a group migrates, the Core of the group notifies the GNS to update the corresponding (group, Core) entry.

A. Joining Process

When a DR of a LAN, which is not on the multicast tree, receives a request (by running IGMP, for example) for a group, it invokes the Joining Process by sending a JOIN-REQUEST hop-by-hop toward the Core of the requested group to build its upstream branch of the tree. A DR that originates a JOIN-REQUEST for a group is called a *Join Pending Router (JPR)*. The first on-tree DR that the JOIN-REQUEST message encounters replies with an acknowledgement (JOIN-ACK) to the JPR, after which it forwards the JOIN-REQUEST to its upstream Agent. When the Agent of the requested group receives the JOIN-REQUEST, it updates its *MemT*. The JPR joins the group and is called a member DR after receiving the JOIN-ACK.

JOIN-REQUEST forces any off-tree DR on the path from the JPR to the Core to forward the request and prepare to join the tree. The JOIN-ACK traverses in reverse the path of the corresponding join message and forces those off-tree DRs to become the tree node. They are then called non-member DRs.

If a non-member DR that is already on the tree receives a request (by running IGMP, for example) for the group, it also invokes the Joining Process and becomes a member DR, but it will not receive any acknowledgement because its upstream branch has already been created. Once the upstream Agent receives the JOIN-REQUEST, it also records the JPR information in its *MemT*.

Message Content: JOIN-REQUEST(*group, count*)
JOIN-ACK(*group*)

In SCMP, a JOIN-REQUEST does not have to reach the Core, only an Agent on the existing tree. To update the Agent's *MemT*, a JOIN-REQUEST carries an extra variable, "*count*," that contains the path delay from JPR to the first Agent that the message encounters. On the path from a JPR to the Core, each on-tree or off-tree DR must add $dist(\textit{itself}, PreHop(\textit{JOIN-REQUEST}))$ to the *count*. Initially, *count* is zero. The following table describes the Joining Process.

Table 1. Joining process.

<p><u>JPR Rule</u> <i>msg(JOIN-REQUEST).count := 0;</i> <i>send the msg(JOIN-REQUEST) toward group's Core;</i> if <i>itself is an off-tree DR originally</i> then <i>create a new MRT-entry for the group after receiving a msg(JOIN-ACK);</i></p> <p><u>Off-Tree DR Rule</u> if <i>receive msg(JOIN-REQUEST)</i> then <i>msg.count = msg.count + dist(itself, PreHop(msg));</i> <i>forward the msg toward the group's Core;</i> if <i>receive msg(JOIN-ACK)</i> then <i>create a new MRT-entry for the group;</i> <i>forward the msg to the JPR;</i></p> <p><u>On-Tree DR Rule</u> if <i>receive msg(JOIN-REQUEST)</i> then <i>msg.count = msg.count + dist(itself, PreHop(msg));</i> <i>forward the msg to its parent;</i> if <i>PreHop(msg) is an off-tree DR</i> then <i>reply JOIN-ACK to the JPR, add PreHop(msg) to its child-list of MRT-entry;</i></p> <p><u>Agent Rule</u> if <i>receive msg(JOIN-REQUEST)</i> then <i>add a new entry (JPR, msg.count + dist(itself, PreHop(msg))) to its MemT;</i> if <i>PreHop(msg) is an off-tree DR</i> then <i>reply JOIN-ACK to the JPR, add PreHop(msg) to its child-list of MRT-entry;</i></p>
--

B. Leaving Process

After all the attached members of a DR have left the group, the DR takes one of the following three actions: (1) It leaves the group and becomes a non-member DR but still is on the tree if it is not a leaf node. (2) It notifies the GNS that the group is terminated if the DR is the Core and is a leaf node. (3) It leaves the group by sending a LEAVE-REQUEST message to its parent if the DR is a leaf node but not the Core. A DR that originates a LEAVE-REQUEST for a group is called a *Leave Pending Router (LPR)*.

LEAVE-REQUEST forces any on-tree node on the path from the LPR to its upstream Agent to forward the request. If a non-member DR on the path becomes a leaf after it deals with the request, it must leave the tree by sending a PRUNE message to its parent to prune its upstream link. In SCMP, a LEAVE-REQUEST or PRUNE will not be acknowledged.

Message Content: LEAVE-REQUEST(*group, isAgent*)
 PRUNE(*group*)

If LPR is an Agent, it sets the Boolean value of “*isAgent*” of LEAVE-REQUEST to true, and the message must reach the Core; otherwise, the message will only reach the Agent of the LPR. Table 2 describes the Leaving Process.

Table 2. Leaving process.

<p><u>LPR Rule</u> <i>if itself is an Agent then</i> $msg.isAgent := true$; <i>send the msg(LEAVE-REQUEST) to its parent, delete the MRT-entry;</i></p> <p><u>On-Tree DR Rule</u> <i>if receive msg(LEAVE-REQUEST) then</i> <i>forward the msg to its parent;</i> <i>if LPR is its child then</i> <i>remove the LPR from its child-list of the MRT-entry;</i> <i>if itself is a non-member DR and its child-list of the MRT-entry is empty then</i> <i>send a msg(PRUNE) to its parent, delete the MRT-entry;</i></p> <p><i>if receive msg(PRUNE) then</i> <i>remove PreHop(msg) from its child-list of the MRT-entry;</i> <i>if itself is a non-member DR and its child-list of the MRT-entry is empty then</i> <i>send a msg(PRUNE) to its parent, delete the MRT-entry;</i></p> <p><u>Agent Rule</u> <i>if receive msg(LEAVE-REQUEST) then</i> <i>if LPR is its child then</i> <i>remove the LPR from its child-list of the MRT-entry;</i> <i>if LPR is an Agent then</i> <i>if itself is the Core then</i> <i>remove the LPR from the AL;</i> <i>else forward the msg toward the Core;</i> <i>else remove the LPR from its MemT;</i></p>

3.2 Expanding and Shrinking Agents

In SCMP, an Agent is responsible for maintaining DRs where the path delays between them do not exceed the maximum path delay bound δ . In other words, the depth of a subtree from the root to its farthest leaf can not be greater than δ . A multicast tree will be expanded or shrunk when its membership changes. We must adjust the number of Agents to maintain the DRs in these subtrees so that the depth of each subtree will not be greater than δ .

A. Expanding Agents Process

If the depth of a subtree is larger than δ , the Agent must create at least one Agent to keep the path delay between the Agent and the DRs within the delay bound. In Fig. 2, δ is 3, the depth of subtree 0 is 5. Therefore the gray DRs are out of bounds. We create additional Agents, Agent 1 for subtree 1, to ensure that the depth of each subtree does not exceed the bound.

Periodically, each Agent checks its depth (the maximal path delay in its *MemT*). If its depth is larger than δ , it multicasts a CREATE-AGENT-REQUEST message to its children. Once a non-leaf DR receives a CREATE-AGENT-REQUEST from its parent, it also multicasts the message to its children. In this way, these messages eventually reach all the leaves of this subtree. When a leaf or an adjacent downstream Agent whose parent is a subtree leaf receives a CREATE-AGENT-REQUEST, it sends a DISCOVER to its parent. After a DISCOVER passes through δ , the last DR in which the DISCOVER stays is chosen as a new Agent.

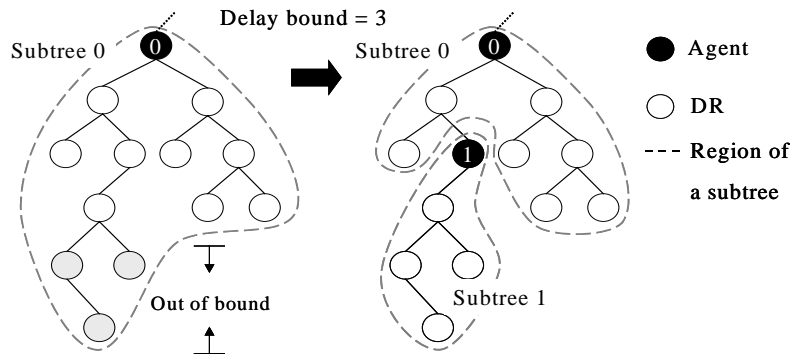


Fig. 2. An example of expanding agents.

Message content: CREATE-AGENT-REQUEST(*group*)
DISCOVER(*group*, *remain-distance*, *DR-list*)

In SCMP, for discovering new Agents and updating the Agents' *MemT*, the "*remain-distance*" of a DISCOVER is the remainder distance that a DISCOVER can traverse. Each element in the "*DR-list*" of a DISCOVER contains a pair (*DR*, *dist*), where "*DR*" is a DR that the DISCOVER has traversed, and the "*dist*" records the distance between the DR and the Agent that receives the DISCOVER message. Initially, the *DR-list* is empty. Table 3 describes the Expanding Agents Process.

B. Shrinking Agents Process

If an Agent detects that the path delay from it to each DR of a downstream subtree rooted at some adjacent downstream Agent does not exceed δ , the Agent can merge this downstream subtree. In Fig. 3, δ is 3, but the maximum path delay from Agent 0 to each DR of subtree 1 does not exceed δ . Thus, subtree 0 and subtree 1 can be merged into one larger subtree, and the redundant Agent 1 can be removed.

In SCMP, when an Agent is created or removed, it must send an AGENT-UPDATE to the Core of the group to update the Core's *AL*. When an Agent executes the Expanding or Shrinking Agents Process, it sets its state to *active*. After the Agent finishes the expanding or shrinking process, it sets its state to *passive*. While an Agent is in the active state, it rejects any REMOVE-AGENT-REQUEST sent by its upstream Agent.

Periodically, each Agent multicasts a REMOVE-AGENT-REQUEST message to its children. Once a non-leaf DR (intermediate DR) receives a REMOVE-AGENT-REQUEST from its parent, it also multicasts the message to its children. In this way, these messages reach all the adjacent downstream Agents of the requesting Agent. When an Agent receives a REMOVE-AGENT-REQUEST from the upstream Agent, it replies with a REJECT-REMOVE-ACK if its state is active; otherwise, it checks the path delay between the requesting Agent and its farthest leaf. If this path delay $> \delta$, the Agent replies with a REJECT-REMOVE-ACK. If this path delay $\leq \delta$, the Agent becomes a non-Agent DR and sends ACCEPT-REMOVE-ACK along with its *MemT* to the requesting Agent.

Table 3. Expanding agents process.

<p><u>Requesting Agent Rule</u> <i>if</i> state = passive and its depth $> \delta$ <i>then</i> state := active, clear MemT; multicast msg(CREATE-AGENT-REQUEST) to its children; wait until msg(DISCOVER) is received from all its children and then update its MemT; /* After receiving a DISCOVER, the Agent first adds the distance between itself and the Prehop(DISCOVER) to the dist of each pair in the DR-list and then appends all the pairs (DR, dist) to its MemT.*/ state := passive;</p> <p><u>Adjacent Downstream Agent Rule</u> <i>if</i> receive msg(CREATE-AGENT-REQUEST) <i>then</i> create a msg(DISCOVER), msg.remain-distance := δ, msg.DR-list := ϕ; send the msg(DISCOVER) to its parent;</p> <p><u>Leaf Rule</u> <i>if</i> receive msg(CREATE-AGENT-REQUEST) <i>then</i> create a msg(DISCOVER), msg.remain-distance := $\delta - \text{dist}(\text{itself}, \text{parent})$; msg.DR-list := (itself, 0), send the msg(DISCOVER) to its parent;</p> <p><u>Non-Leaf DR Rule</u> <i>if</i> receive msg(CREATE-AGENT-REQUEST) <i>then</i> multicast the msg to its children; <i>if</i> receive msg(DISCOVER) <i>then</i> msg.remain-distance := msg.remain-distance - dist(itself, parent); update the msg.DR-list; /*add dist(itself, prehop(msg)) to the dist of each pair of the DR-list*/ <i>if</i> msg.remain-distance < 0 <i>then</i> become a new Agent, its MemT := msg.DR-list \cup received msg.DR-list; create a new msg(DISCOVER), msg.remain-distance := δ, msg.DR-list := ϕ; send the msg(DISCOVER) to its parent; send an AGENT-UPDATE message to the Core; wait until msg(DISCOVER) is received from all its children and then update its MemT; <i>else if</i> all its children have replied <i>then</i> create a new msg(DISCOVER); msg.remain-distance := Min(received msg.remain-distance); msg.DR-list := \cup(received msg.DR-list); <i>if</i> itself is a member DR <i>then</i> add (itself, 0) to the tail of msg.DR-list; send the msg(DISCOVER) to its parent;</p>

In this way, the requesting Agent eventually receives all the replies from its children. If the reply is a REJECT-REMOVE-ACK, the request is rejected. If the reply is an ACCEPT-REMOVE-ACK, the Agent adds new information of the merged downstream subtree to its MemT.

Message content: REMOVE-AGENT-REQUEST(group, distance)
ACCEPT-REMOVE-ACK(group, MemT)
REJECT-REMOVE-ACK(group)

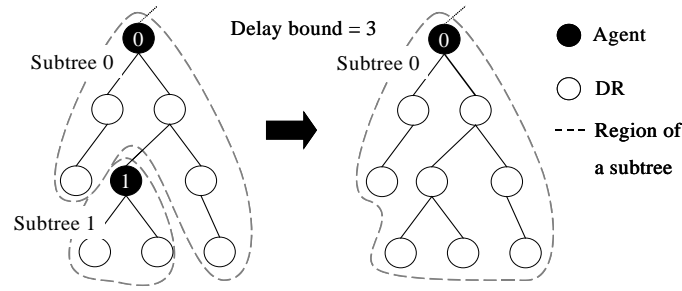


Fig. 3. Shrinking agents example.

In SCMP, a REMOVE-AGENT-REQUEST carries a variable “*distance*” that records the distance of the path between the requesting Agent and one of its downstream Agents. Initially, the “*distance*” is zero. On this path, each intermediate DR must add $dist(\text{itself}, \text{parent})$ to the *distance*. “*MemT*” of ACCEPT-REMOVE-ACK records the *MemT* of the removed adjacent downstream Agent. Table 4 describes the Shrinking Agents Process.

3.3 Core Migrating

Core migrating enables the group to respond to significant changes in membership. In SCMP, we use a distributed algorithm to probe an Agent for use as a new Core.

A. Probing Core Process

First, the current Core is chosen as a probing Agent. The probing Agent first executes the weight calculation process, which will be described in the next section, to get its own weight and an agent-distance-list (*ADL*), where $ADL = \langle (a_1, d_1), (a_2, d_2) \dots (a_n, d_n) \rangle$, a_i is a group agent and d_i is the distance between the probing agent and a_i . The probing Agent sends *ADL* to its adjacent Agents to ask for their weights. Each adjacent Agent calculates its own weight according to the weight function (described in section 3.4) and replies with the result. If the probing Agent’s own weight is lower than the lowest adjacent Agent’s weight, it will stop probing, and that agent will become the new Core. Otherwise, the probing Agent will pick an unvisited adjacent Agent with the lowest weight as the next probing Agent.

Message content: WEIGHT-QUERY(*group*, *ADL*)
WEIGHT-REPLY(*group*, *weight-list*)

Each element in “*weight-list*” contains a pair (*Agent*, *weight*), where “*Agent*” is a queried adjacent Agent, and “*weight*” is the weight of the queried adjacent Agent. Table 5 describes the Probing Core Process.

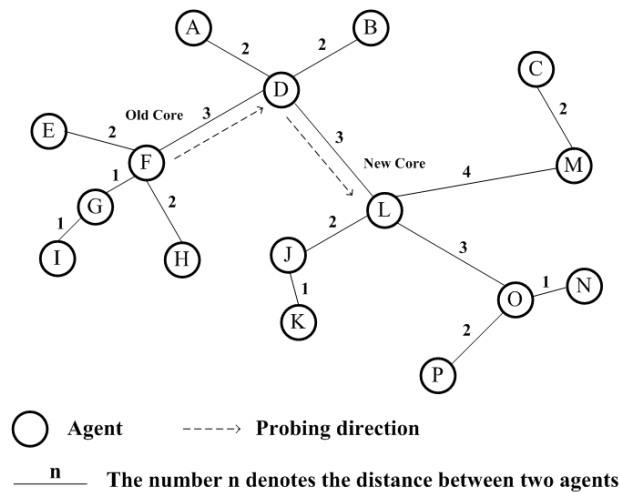
Table 4. Shrinking agents process.

<p><u>Requesting Agent Rule</u> <i>if</i> $state = passive$ <i>then</i> $state := active$, $msg(REMOVE-AGENT-REQUEST).distance := 0$; multicast the $msg(REMOVE-AGENT-REQUEST)$ to its children; repeat <i>if</i> receive $msg(ACCEPT-REMOVE-ACK)$ <i>then</i> update its $MemT$; <i>until</i> all its children have replied; $state := passive$;</p> <p><u>Adjacent downstream Agent Rule</u> <i>if</i> receive $msg(REMOVE-AGENT-REQUEST)$ <i>then</i> $msg.distance := msg.distance + dist(itself, parent)$; <i>if</i> $(state = passive)$ and $(msg.distance + its\ depth) \leq \delta$ <i>then</i> become an non-Agent DR; send an AGENT-UPDATE message to the Core; add $msg.distance$ to the distance of each entry of its $MemT$; create $msg(ACCEPT-REMOVE-ACK)$, $msg.MemT := its\ MemT$; send $msg(ACCEPT-REMOVE-ACK)$ to its parent; else send $msg(REJECT-REMOVE-ACK)$ to its parent;</p> <p><u>Non-Leaf DR Rule</u> <i>if</i> receive $msg(REMOVE-AGENT-REQUEST)$ <i>then</i> $msg.distance := msg.distance + dist(itself, parent)$, multicast the msg to its children; <i>if</i> receive $msg(REJECT-REMOVE-ACK)$ or $msg(ACCEPT-REMOVE-ACK)$ <i>then</i> <i>if</i> all its children have replied <i>then</i> <i>if</i> exist a received $msg(ACCEPT-REMOVE-ACK)$ <i>then</i> create a new $msg(ACCEPT-REMOVE-ACK)$; $msg.MemT := \cup(received\ msg.MemT)$; send the $msg(ACCEPT-REMOVE-ACK)$ to its parent; <i>else</i> send a $msg(REJECT-REMOVE-ACK)$ to its parent;</p> <p><u>Leaf Rule</u> <i>if</i> receive $msg(REMOVE-AGENT-REQUEST)$ <i>then</i> send a $msg(REJECT-REMOVE-ACK)$ to its parent;</p>

The network shown in Fig. 4 illustrates Probing Core Process. First, the current Core, Agent F is chosen as the probing Agent. Probing Agent F queries the weights of Agent D, E, G and H, and then it picks Agent D, whose weight is minimal, as the next probing Agent. Later, probing Agent D picks Agent L as the next probing Agent in the same way. After querying the weights of Agent J, O and M, probing Agent L finds its own weight is lowest, and it becomes the new group's Core.

Table 5. Probing core process.

<p>Probing Agent Rule <i>invoke Calculating Weight Process (described in Table 6) to get its weight and ADL;</i> <i>multicast msg(WEIGHT-QUERY) to its children;</i> <i>wait until msg(WEIGHT-REPLY) is received from all its children;</i> if its own weight < lowest adjacent Agent weight then invoke Changing Core Process(described in Table 8); else <i>pick an adjacent Agent with the lowest weight to be the next probing Agent;</i> <i>send AL to the next probing Agent;</i></p> <p>Adjacent Agent Rule if receive msg(WEIGHT-QUERY) then <i>invoke Calculating Weight Process (described in Table 7);</i> <i>create a msg(WEIGHT-REPLY), msg.weight-list := (itself, its weight);</i> <i>send the msg(WEIGHT-REPLY) to its parent;</i></p> <p>Non-Leaf DR Rule if receive msg(WEIGHT-QUERY) then multicast the msg to its children; if receive msg(WEIGHT-REPLY) and all its children have replied then <i>create a new msg(WEIGHT-REPLY);</i> <i>msg.weight-list := \cup(received msg.weight-list);</i> <i>send the msg(WEIGHT-REPLY) to its parent;</i></p> <p>Leaf Rule if receive msg(WEIGHT-QUERY) then <i>msg(WEIGHT-REPLY).weight-list := ϕ, send the msg to its parent;</i></p>



Agent	DelayVar
A	9
B	9
C	12
D	7
E	12
F	11
G	12
H	12
I	13
J	9
K	10
L	6
M	10
N	11
O	10
P	11

Fig. 4. An example of the probing core process using the delay variation (*DelayVar*) weight function.

B. Calculating Weight Process

When a probing Agent wants to calculate its own weight, it multicasts a query message to the group's tree, and reply messages come back from all of the leaves to the probing Agent. These replies also record the Agents they traverse and the distances between these traversed Agents and the probing Agent. Therefore, the probing Agent can compute its own weight and get the ADL by means of the replies. Table 6 describes the Calculating Weight Process of the Probing Agent.

Message content: DISTANCE-QUERY(*group*)
DISTANCE-REPLY(*group*, *ADL*)

Table 6. Calculating weight process of the probing agent.

<p>Probing Agent Rule <i>msg</i>(DISTANCE-QUERY).<i>adjacent</i> := true, multicast the message to all its children; wait until <i>msg</i>(DISTANCE-REPLY) is received from all its children; <i>ADL</i> := \cup(received <i>msg</i>.<i>ADL</i>); compute its own weight using the function <i>DelayVar</i> defined in section 3.4;</p> <p>Other Agent Rule if receive <i>msg</i>(DISTANCE-QUERY) then multicast the <i>msg</i> to its children; if receive <i>msg</i>(DISTANCE-REPLY) then add <i>dist</i>(<i>itself</i>, <i>PreHop</i>(<i>msg</i>)) to the <i>d_i</i> of each pair of <i>msg</i>.<i>ADL</i>; if all children have replied <i>msg</i>(DISTANCE-REPLY) then <i>temp_ADL</i> := (<i>itself</i>, 0) \cup(received <i>msg</i>.<i>ADL</i>); create a <i>msg</i>(DISTANCE-REPLY), <i>msg</i>.<i>ADL</i> := <i>temp_ADL</i>; send the <i>msg</i> to its parent;</p> <p>Non-Leaf DR Rule if receive <i>msg</i>(DISTANCE-QUERY) then multicast the <i>msg</i> to its children; if receive <i>msg</i>(DISTANCE-REPLY) then add <i>dist</i>(<i>itself</i>, <i>PreHop</i>(<i>msg</i>)) to the <i>d_i</i> of each pair of <i>msg</i>.<i>ADL</i>; if all children have replied <i>msg</i>(DISTANCE-REPLY) then <i>temp_ADL</i> := \cup(received <i>msg</i>.<i>ADL</i>); create a <i>msg</i>(DISTANCE-REPLY), <i>msg</i>.<i>ADL</i> := <i>temp_ADL</i>; send the <i>msg</i> to its parent;</p> <p>Leaf Rule if receive <i>msg</i>(DISTANCE-QUERY) then <i>msg</i>(DISTANCE-REPLY).<i>ADL</i> := ϕ, send the <i>msg</i> to its parent;</p>
--

When an adjacent Agent is asked to calculate its own weight, it does so using its own *temp_ADL* and the *ADL* of the message WEIGHT-QUERY that the probing Agent sends out. Table 7 describes the Calculating Weight Process of the Adjacent Agent.

As shown in Fig. 4, if the probing Agent is Agent D, it would multicast DISTANCE-QUERY to the group's tree. It's *ADL* is {(A, 2), (B, 2), (C, 9), (D, 0), (E, 5), (F, 3), (G, 4), (H, 5), (I, 5), (J, 5), (K, 6), (L, 3), (M, 7), (N, 7), (O, 6), (P, 8)} after all of the DISTANCE-REPLYs are received. Its adjacent Agents also collect the partial *ADL*

Table 7. Calculating weight process of the adjacent agent.**Adjacent Agent Rule**

$ADL := temp_ADL \cup \{ (a_i, d_i + dist(itself, Probing\ Agent)) \mid a_i \in WEIGHT_QUERY_ADL \text{ and } a_i \notin temp_ADL \};$

compute its own weight using the function DelayVar defined in section 3.4;

information in $temp_ADL$. For example, Agent L's $temp_ADL$ is $\{(C, 6), (J, 2), (K, 3), (L, 0), (M, 4), (N, 4), (O, 3), (P, 5)\}$ after the DISTANCE-REPLYs are received. When Agent L is asked to calculate its own weight, it updates its ADL to $\{(C, 6), (J, 2), (K, 3), (L, 0), (M, 4), (N, 4), (O, 3), (P, 5)\} \cup \{(A, 2 + 3), (B, 2 + 3), (D, 0 + 3), (E, 5 + 3), (F, 3 + 3), (G, 4 + 3), (H, 5 + 3), (I, 5 + 3)\}$, where 3 is the $dist(D, L)$.

C. Changing Core Process

The final probing Agent sends a CHANGE-CORE-REQUEST through the tree links to the current Core. After receiving the request, the current Core becomes a non-Core Agent and replies with a CHANGE-CORE-REPLY through the reverse path of the corresponding CHANGE-CORE-REQUEST message to the new Core. When the requesting Agent receives this reply, it becomes the new Core. Note that each node in the path of the CHANGE-CORE-REPLY message modifies its MRT -entry to maintain the correct parent-child relationship. Table 8 describes the Changing Core Process.

Message content: CHANGE-CORE-REQUEST($group$)
CHANGE-CORE-REPLY($group$)

3.4 Weight Functions

In SCMP, we use a weight function to find an optimal Core based on delay variation ($DelayVar$). This metric is computed as follows:

$$DelayVar = \max_{a, b \in S, a \neq b} \{ |dist(root, a) - dist(root, b)| \}$$

S is the set of Agents recorded in the Agent list (AL), and $root$ is the node that is computing its own weight. We also can use other performance metrics to find an optimal Core, such as: (1) the average end-to-end delay ($AvgDist$) [13], (2) the maximum end-to-end delay ($MaxDist$), and (3) the maximum diameter ($MaxDiam$) [5]. The metrics are computed as follows:

$$AvgDist = \frac{1}{|S|} \sum_{a \in S} dist(root, a),$$

$$MaxDist = \max_{a \in S} dist(root, a),$$

$$MaxDiam = \max_{a \in S} dist(root, a) + \max_{b \in S, b \neq a} dist(root, b).$$

Table 8. Changing core process.

<p><u>Final Probing Agent Rule</u> <i>send msg(CHANGE-CORE-REQUEST) to its parent;</i> <i>if receive msg(CHANGE-CORE-REPLY) then</i> <i>add PreHop(msg) to its child-list of MRT-entry;</i> <i>MRT-entry.p := null;</i> <i>become the group's new Core;</i> <i>send an AGENT-UPDATE message to the Core;</i></p> <p><u>Current Core Rule</u> <i>if receive msg(CHANGE-CORE-REQUEST) then</i> <i>delete PreHop(msg) from its child-list of MRT-entry;</i> <i>MRT-entry.p := PreHop(msg);</i> <i>send msg(CHANGE-CORE-REPLY) to its parent, become a non-Core Agent;</i></p> <p><u>Intermediate Agent and DR Rule</u> <i>if receive msg(CHANGE-CORE-REQUEST) then</i> <i>temp = PreHop(msg), forward the msg to its parent;</i> <i>if receive msg(CHANGE-CORE-REPLY) then</i> <i>add PreHop(msg) to its child-list of MRT-entry;</i> <i>delete temp from its child-list of MRT-entry;</i> <i>MRT-entry.p := temp;</i> <i>forward the msg to its parent;</i></p>

4. PERFORMANCE EVALUATION

In this section, we will analyze the simulation results for the Random Core Based Tree (RCBT), SCMP, and Distributed Center-Location Algorithm (DCLA) [10]. In the RCBT heuristic, the Core is chosen randomly among the sources and never moves. This approach is also equivalent to selecting the first source or the initiator of the multicast group, as suggested by PIM [3] and CBT [2].

The simulation was written in Java and executed on Microsoft Windows NT.

4.1 Generating Random Graphs

The simulation generates almost real-world networks first and then executes the protocols mentioned above. We use the random graph model presented by Waxman [15] to generate networks. The probability that an edge exists between any two nodes u and v is given by the following probability function:

$$P(u, v) = \beta \exp \frac{-d(u, v)}{L\alpha},$$

where $d(u, v)$ is the distance between the two nodes, L is the maximum possible distance, and α and β are adjustable parameters in the range $0 < \alpha, \beta \leq 1$. These adjustable parameters affect the connectivity and the degree of the graph formed. Larger values of α increase the ratio of longer edges to shorter edges, while larger values of β increase the average node degree.

4.2 Dynamic Group Membership

In this paper, we assume that each group member can be a sender and a receiver. Thus, the number of sources equals the number of members. A node joins or leaves a group using the probability function: $P_c(k) = \frac{\gamma(n-k)}{\gamma(n-k)+(1-\gamma)k}$, where k is the current number of group members, n is the number of nodes in the network, and γ is a parameter in the range $0 < \gamma < 1$ and is the mean fraction of nodes that are members at equilibrium. If $\gamma = \frac{k}{n}$, then $P_c(k) = \frac{1}{2}$; if $\gamma > \frac{k}{n}$, then $P_c(k) > \frac{1}{2}$; and if $\gamma < \frac{k}{n}$, then $P_c(k) < \frac{1}{2}$.

4.3 Simulation Results

The simulation was performed ten times for a given network and each protocol. The group generated consisted of 400 nodes and average node degree was 5.

A. Tree Cost, Maximum Delay and Delay Variation

A lower tree cost reduces the overall bandwidth requirements and effectively raises the number of multicast groups that can be supported by the network. This is especially important since it is expected that the number of multicast groups will become very large in the future. A lower maximum delay means that packets sent by sources will tend to arrive at their destination sooner. A lower delay variation enables synchronization among the various receivers and issues that no receiver is “left behind” and that none is “far ahead” during the lifetime of the session.

We compared these metrics of SCMP and DCLA with those of RCBT and simulated SCMP and DCLA using the weight function DelayVar, AvgDist, MaxDist, and MaxDiam (defined in section 3.4). In SCMP, we let the maximum delay bound δ of a subtree be $1/6$ of the diameter of the initial given network. In Figs. 5, 6 and 7, the Y-axis plots the ratio between the average Actual Cost, Maximum Delay, and Delay Variation at a center point located using each weight function and RCBT, and the X-axis plots the percentage of group membership.

As shown in Figs. 5, 6, and 7, the weight functions that typically lead to the best actual cost result in the worst maximum delay and delay variation, showing that a cost versus delay tradeoff exists. The AvgDist function leads to a worse maximum delay and delay variation than do DelayVar, MaxDist and MaxDiam. This is because AvgDist yields a lower average delay and may yield a higher maximum delay and maximum delay variation. We can also see that the DelayVar weight function gives the best delay variation. Therefore, we can use this weight function to minimize the delay variation.

Figs. 6 and 7 show that the number of group members is small when a group is initiated, so the differences of the maximum delay and delay variation between a fixed Core and migrating Core are slight. When the number of group's members increases, this difference becomes large.

Depending on the simulation results, DCLA had the best delay variation, actual tree cost, maximum delay, and delay variation. SCMP also had good performance, only 2% worse than the percentage measured for DCLA. This is because the Core of SCMP has the minimum weight among all of the Agents rather than all of the members.

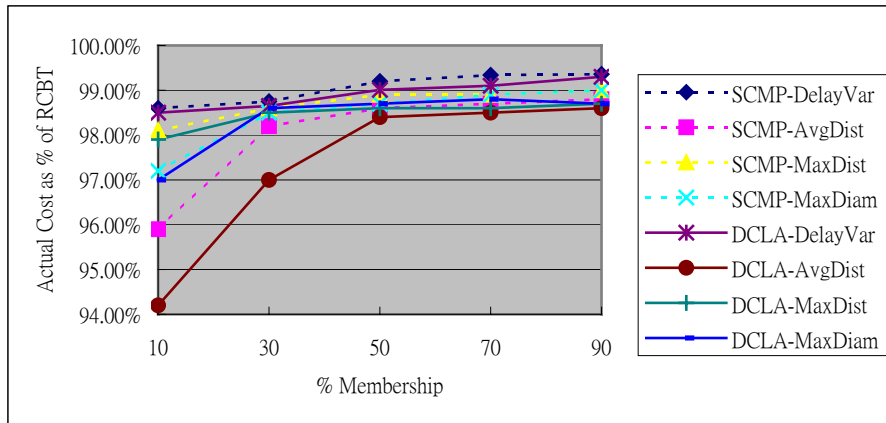


Fig. 5. Actual cost of the resulting multicast tree.

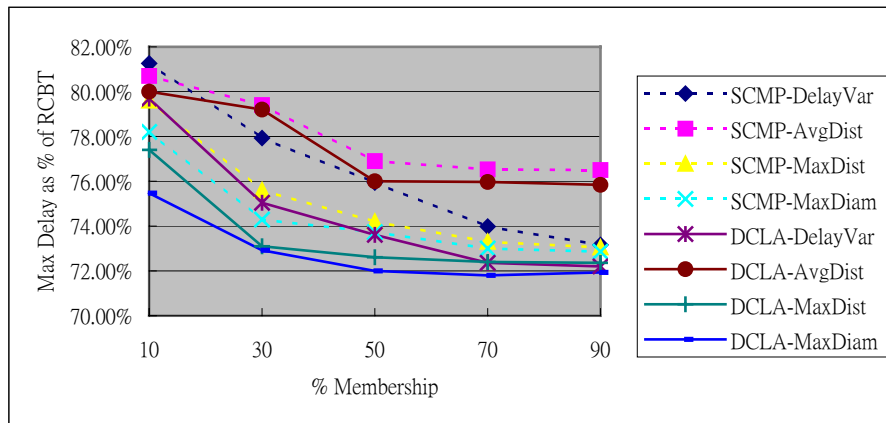


Fig. 6. Maximum delay of the resulting multicast tree.

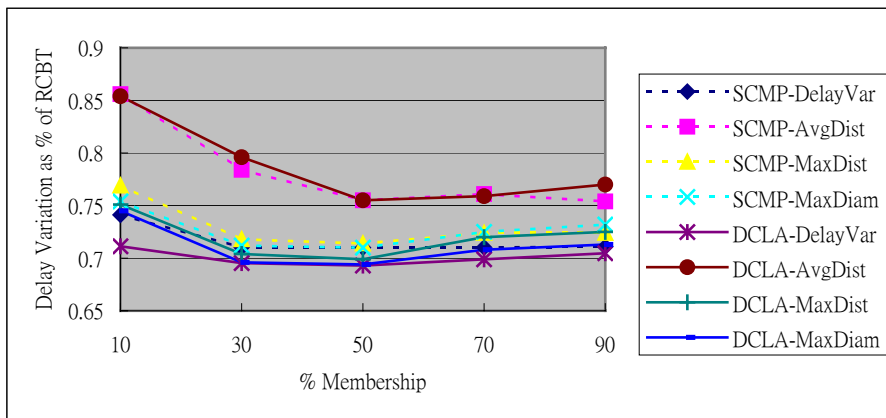


Fig. 7. Delay variation of the resulting multicast tree.

B. State Space of Core Required

Although knowledge of the network topology is not explicitly needed by DCLA [10], some knowledge, such as the source-list/member-list, is necessary to compute the metrics to migrate the Core to the best location. If the number of group's members increases, the scalability problem will occur. Fig. 8 shows the state space of the Core required for DCLA and SCMP ($\delta = 1/3$ and $1/6$ the diameter of the network). The Y-axis plots the information of the members/Agents that the Core in DCLA/SCMP should store.

Compared to DCLA, the state space of the Core in SCMP is 85%~65% smaller. This is because the Core must maintain information of all of the members in DCLA, but the Core only maintains information of all of the Agents in SCMP.

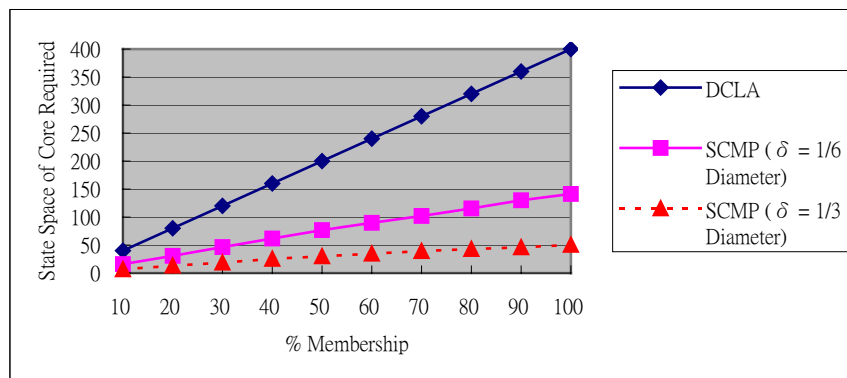


Fig. 8. State space of Core required for the resulting multicast tree.

C. Overhead of Core Migration

In [7-10], to find a new Core, each probing node and the neighboring node should compute their weights by querying all of the members/sources. Overhead messages occurred by these querying and replying actions (described in section 3.4) will cause traffic congestion. The long runtime for getting weights will slow down Core migration.

The transition from the probing node starting process to picking the next probing node is defined as the "probing step." Fig. 9 shows the message overhead during the probing step. The Y-axis plots the number of messages of SCMP and DCLA. The Y-axis plots the number of messages incurred during a probing step. SCMP had a lower message overhead than DCLA did because only Agents are queried to compute weights in SCMP. Fig. 10 shows that the runtime overhead decreased during the probing step for DCLA compared to SCMP. The line indicates the ratio of the runtime overhead of Core Migrating using SCMP to that using DCLA. SCMP had a lower runtime overhead than DCLA did, and the runtime overhead decreased faster as the number of group' members increased. This is because the only data needed to compute weights is the information of the Agents.

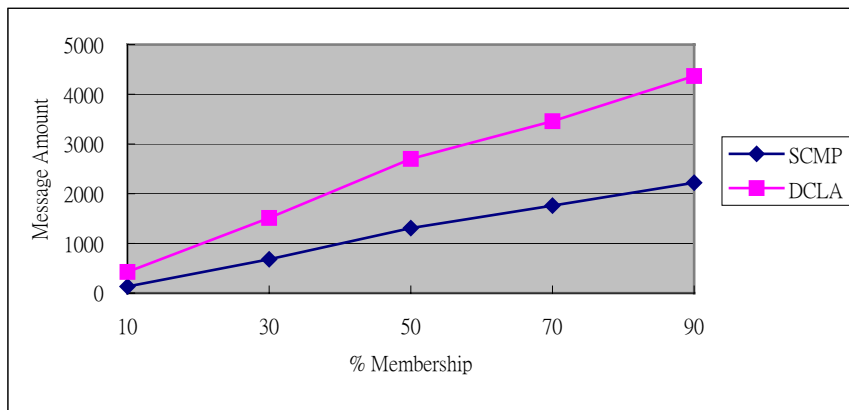


Fig. 9. Message overhead of the Core migrating process.

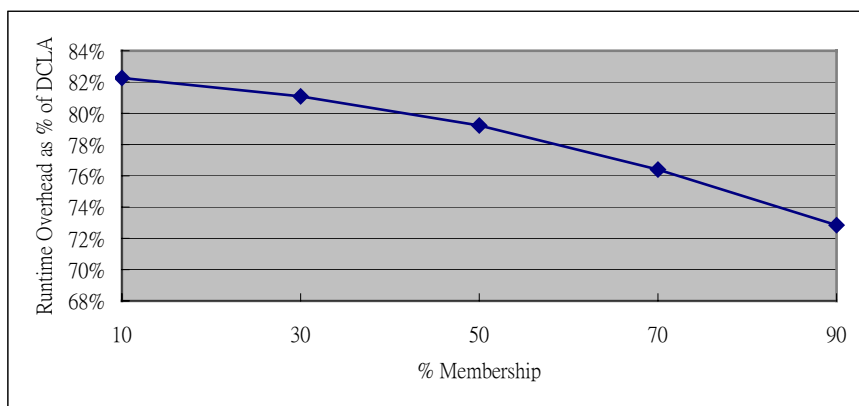


Fig. 10. Runtime overhead of the Core migrating process.

5. CONCLUSIONS

In this paper, we have presented a scalable distributed protocol that can be used to migrate the Core to a near-optimal location in a dynamic multicast tree, and that allows the Core to migrate efficiently when the multicast tree is expanded or shrunken. In SCMP, information of all of the members is distributed among local Agents; the Core only maintains information of the Agents of the group. Also, only the Agents participate in Core selection. Therefore, the proposed protocol reduces the runtime overhead and message overhead while computing Core migration.

Simulations of RCBT, SCMP, and DCLA [10] show that the migrating Core heuristic achieves better performance than the fixed Core heuristic. DCLA has a better actual tree cost, maximum delay, and delay variation metric than SCMP does, but the difference is slight. On average, the state space of Core required in SCMP is much less than that in DCLA. Moreover, the runtime overhead and message overhead of SCMP are obviously lower than those of DCLA.

Based on SCMP, our future work will focus on supporting the fault-tolerant capacity and cooperation with other CBT protocols in multicast networks. We will investigate other Core migration methods other than the probing approach used in SCMP.

REFERENCES

1. L. C. Wu, L. S. Lin, and S. C. Shiao, "Constructing delay-bounded multicast tree in computer networks," *Journal of Information Science and Engineering*, Vol. 17, 2001, pp. 507-524.
2. A. J. Ballardie, et al., "Core based trees," in *Proceedings of ACM SIGCOMM*, 1993, pp. 85-95.
3. S. Deering, et al., "The PIM architecture for wide-area multicast routing," *Networking, IEEE/ACM Transactions*, Vol. 4, 1996, pp. 153-162.
4. K. L. Calvert, E. W. Zegura, and M. J. Donahoo, "Core selection methods for multicast routing," in *Proceedings of 4th International Conference on Computer Communications and Networks, IEEE*, 1995, pp. 638-642.
5. D. W. Wall, "Mechanisms for broadcast and selective broadcast," Ph.D. Dissertation, Stanford University, 1980.
6. C. Liu, M. J. Lee, and T. N. Saadawi, "Core-manager based scalable multicast routing," in *Proceedings of International Conference on Communication, IEEE*, 1998, pp. 1202-1207.
7. S. Ali and A. Khokhar, "Distributed center location algorithm for fault-tolerant multicast in wide-area networks," *17th Proceedings of Reliable Distributed Systems, IEEE*, 1998, pp. 324-329.
8. M. J. Donahoo and E. W. Zegura, "Core migration for dynamic multicast routing," in *Proceedings of 5th International Conference on Computer Communications and Networks, IEEE*, 1996, pp. 92-98.
9. A. Gulati and S. Rai, "Core discovery in internet multicast routing protocol," *Performance, Computing and Communications Conference, 1999 IEEE International*, 1999, pp. 143-149.
10. D. G. Thaler and C. V. Ravishankar, "Distributed center-location algorithms," *IEEE Select. Areas in Journal of Commun.*, Vol. 15, 1997, pp. 291-303.
11. G. N. Rouskas and I. Baldine, "Multicast routing with end-to-end delay and delay variation constraints," *IEEE Select. Areas in Journal of Commun.*, Vol. 15, 1997, pp. 346-356.
12. S. B. Shukla, E. B. Boyer, and J. E. Klinker, "Multicast tree construction in network topologies with asymmetric link loads," Naval Postgraduate School, Technical Report NPS-EC-94-012, 1994.
13. L. Wei and D. Estrin, "A comparison of multicast trees and algorithms," Computer Science Dept., University of Southern California, Technical Report USC-CS-95-560, 1993.
14. R. Voigt, R. Barton, and S. B. Shukla, "A tool for configuring multicast data distribution over global networks," in *Proceedings of International Conference on Networking*, 1995, pp. 455-463.
15. B. M. Waxman, "Routing of multipoint connections," *IEEE Select. Areas in Journal of Commun.*, Vol. SAC-6, 1988, pp. 1617-1622.

16. T. Billhartz, et al., "Performance and resource cost comparisons for the CBT and PIM multicast routing protocols," *IEEE Select. Areas in Journal of Commun.*, Vol. 15, 1997, pp. 304-315.



Ting-Yuan Wang (王鼎允) is a software engineer at the Industrial Technology Research Institute (ITRI) in Taiwan. He received the M. S. degree in Computer Science from Tsing Hua University, Taiwan, in 2000. His research areas are multicast protocol, voice over internet protocol, voice over wireless LAN, and embedded system.



Lih-Chyau Wu (伍麗樵) is an associate professor of the Electrical Engineering Department at National Yunlin University of Science & Technology, Taiwan. She received her B.S. degree from the Department of Information Engineering, National Taiwan University (Taipei, Taiwan), in 1982, and her Ph.D. degree from the Department of Computer Science, National Tsing Hua University (Hsinchu, Taiwan), in 1994. Her research interests include multicast routing, network security, network management and self-stabilization.



Shing-Tsaan Huang (黃興燦) is Professor and Dean of the College of Electrical Engineering and Computer Science, National Central University, Taiwan. His research interests include operating systems, parallel and distributed computing, and fault tolerance computing. He received the Ph.D. degree from the Department of Computer Science, the University of Maryland at College Park. Dr. Huang is an IEEE Fellow for his contributions to parallel and distributed computing.