Protocol Design for Scalable and Adaptive Multicast for Group Communications

De-Nian Yang and Wanjiun Liao
Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan
dnyang,wjliao@cc.ee.ntu.edu.tw

Abstract— **Currently, IP multicast and Explicit Multi-Unicast (Xcast) are two approaches for multicast communications. IP multicast is designed for large groups but is not scalable in terms of the group number because every router in a multicast tree needs to store the forwarding state for each group. In contrast, Xcast is designed for small groups and is not scalable in terms of the group size because each packet's header can include only the addresses of a few receivers. Therefore, the two approaches are designed for different scenarios and address different problems. However, the selection of IP multicast or Xcast is left to end users who will choose the corresponding API in the applications. In other words, the scalability of the network relies on the end users, and is not guaranteed by the protocol itself. In this paper, we address the above issues and propose a multicasting protocol that is scalable and adaptive in terms of both group number and group size. We avoid the disadvantages of IP multicast and Xcast by choosing only a few routers to store the forwarding states, and multicast packets are delivered via Xcast among these routers. The main advantage of our protocol is that the assignment of forwarding states is optimal and performed in a fully distributed manner. We show that IP multicast and Xcast are two extreme and special cases of our approach. We prove that the overhead of our protocol is limited. In addition, the assignment of forwarding states in our protocol is adaptive to the dynamic group membership and the change in network topology. Moreover, our protocol is simple and can be extended from existing multicast routing protocols.**

Index Terms— **Multicast, Xcast, scalability**

I. Introduction

IP multicast can be divided in to two categories: traditional IP multicast and source-specific multicast. Traditional IP multicast is based on the host group model [1] and multicast routing protocols [2]–[5]. It associates each multicast group with a class-D IP address, and the address serves as the destination address of data packets for the group. Multicast addresses are allocated dynamically in order to guarantee the global uniqueness of each class-D address [6]. On the contrary, source-specific multicast (SSM) [7] treats a one-to-many connection as a multicast channel. Each multicast channel is associated with a channel identifier including the sender's address and a class-D address. The class-D address is allocated by the sender. Therefore, it is not required to be globally unique.

Both source-specific multicast and traditional IP multicast adopt a shortest-path tree to deliver multicast data. The routing of the shortest-path tree is the union of the shortest paths from the root to all receivers or from all receivers to the root. For source-specific multicast, the root is the sender, and the tree is regarded as a source-based tree. For traditional IP multicast, the root is a router called the core in CBT [4] or the RP in PIM-SM [5], and a tree of this kind is called a shared tree. Each sender first sends data to the root, and then the root relays the data to all receivers.

Each router in traditional IP multicast and source-specific multicast needs to store a forwarding state for each multicast group. The state specifies the adjacent routers in the multicast tree, and the ID of the forwarding state is the channel ID or the group address. Aggregating multicast forwarding states is more difficult than aggregating unicast forwarding states because the destination addresses, i.e., class-D addresses, are not assigned hierarchically and geographically. In addition, the distribution of receivers can be arbitrary and independent of the destination addresses. Therefore, each router in the tree may need a very large memory space to store all multicast forwarding states when there are a larger number of multicast groups in the network. Moreover, it may take a long time to find the forwarding state. The problem is more serious for source-specific multicast because a dedicated tree is required for each source of a group. Therefore, IP multicast is not scalable in terms of the number of multicast groups [8].

In contrast, in Explicit Multi-Unicast (Xcast) [9] each router delivers multicast data based on unicast forwarding states. No router in a multicast tree stores the multicast forwarding states. Xcast includes the addresses of all receivers of a multicast tree in the header of each data packet. When a router in the tree receives a packet, it first finds the next-hop router of each receiver, duplicates the packet to each downstream interface, and then sends the packet to each next-hop router. The header of each duplicated packet contains only the receivers in the sub-tree rooted at the next-hop router. Although multicast forwarding states are not required, Xcast is not suitable for groups with many receivers since the size of the header and the packet processing delay in a router grows as the number of receivers increases [9]. Therefore, Xcast is scalable in terms of the number of groups but is not scalable in terms of the number of receivers in a group. On the contrary, IP multicast is scalable in terms of the number of receivers in a group but is not scalable in terms of the number of groups. Therefore, it is generally believed that both mechanisms are applicable in different scenarios [9].

However, currently the selection of IP multicast or Xcast is left to end users who will choose the corresponding API in the client programs. Since users may not be aware of the scalability problem of the network, the network operators, who desire to avoid the scalability problem, may suffer when end users choose improper schemes for group communications. On the other hand, it is difficult for the network operators to select the scheme for end users of each group because the network operator may not be aware of the application of the group. In addition, a suitable scheme may not even exist because the number of members in a group can vary with time. In this case, a scheme properly chosen by a group initially may not be scalable later, and switching to the other scheme leads to service disruption. Therefore, we need a single multicast protocol scalable in terms of both the number of receivers and the number of groups.

In this paper, we propose a scalable and adaptive multicast protocol for group communications. Our protocol adaptively selects a few on-tree routers to store the multicast forwarding states to achieve scalability, and multicast packets are delivered via Xcast between routers that store the forwarding states. The advantages of our protocols are many-fold.

- Our protocol is scalable in terms of the group size, because the root of the tree does not include the addresses of receivers in the packet header for a large multicast group. Instead, several downstream routers with states are indicated as the destinations of the packets sent via Xcast from the root of the tree.
- Our protocol is scalable in terms of the group number. For small multicast groups, our protocol converges to Xcast, and no router stores the forwarding state of the group. As the group size increases, our protocol adaptively chooses a few routers to store the forwarding states.
- Our protocol obtains the optimal solution. Even though each router is not aware of the topology of the multicast tree, our protocol can find the minimum number of routers storing the state of the group in a fully distributed manner, given the maximum number of addresses that can be included in each Xcast packet. In addition, we prove that the overhead of our protocol is limited.
- The routers with the forwarding states can be either branching or non-branching ones in a multicast tree. We show that previous works [10][11] storing forwarding states in only the branching routers are special cases of our approach. Moreover, we allow the backbone routers with large degrees [16], which tend to be branching routers in a tree, not to store the forwarding states to maintain high forwarding speed.
- Our protocol is adaptive to the dynamic group membership. Our protocol adaptively removes or creates the forwarding states in routers toward the optimal solution as the group number decreases or increases. The routers that need to remove or create the states are correlated to the leaving or joining members.
- Our protocol is simple. It can be extended from existing multicast routing protocols using unidirectional multicast tree, such as PIM-SM. In addition to the joining and leaving operations similar to those in existing multicast routing protocols, we add only a few operations to obtain the optimal assignment of forwarding states among routers. In addition, our protocol can be incrementally deployed in a network that adopts existing multicast routing protocols to gradually increase the scalability.

The rest of this paper is summarized as follows. We describe the details of our protocol in Section II and analyze the overhead in Section III. In Section IV, we discuss the related issues of our protocol, such as incremental deployment based on existing multicast routing protocols. Section V shows the simulation results, and finally, we conclude this paper in Section VI.

II. Protocol Design

A. Protocol Overview

Our protocol adaptively converges toward the optimal assignment of multicast forwarding states among routers. The objective of our protocol is to minimize the number of routers that store the forwarding states of each multicast tree. The number of destinations addresses in each Xcast packet cannot exceed δ (to consider the header size and to limit the forwarding delay in each router), where δ is a network design parameter and can be assigned by network operators. Each router that stores the forwarding state of a multicast group is regarded as a *state node* of the corresponding multicast tree. A downstream router d of the state node u is a *downstream state node* of u if all routers between u and d are *stateless*, i.e., not a state node. In this case, u is the upstream state node of d. A state node can have at most one upstream state node but multiple downstream state nodes. However, a state node can have at most δ downstream state node from each downstream interface because each Xcast packet sent from the interface can include at most δ destination addresses.

Our protocol is simple but can find the optimal assignment. It includes the following two operations: REMOVE and MOVE. Operation REMOVE reduces the number of state nodes, and MOVE packs the state nodes in the tree such that more routers can remove the stored forwarding states later. Specifically, each state node d independently first tries REMOVE, which removes its forwarding state, if the number of destination addresses in each Xcast packet sent from the upstream state node u does not exceed δ. Otherwise, state node d tries MOVE, which moves its forwarding state to the parent node, if the parent node of d is stateless and the number downstream state nodes is no more than δ.

Our protocol is simple but can find the optimal assignment. It includes the following two operations: REMOVE and MOVE. Operation REMOVE reduces the number of state nodes, and MOVE packs the state nodes in the tree such that more routers

can remove the stored forwarding states later. Specifically, each state node *d* independently first tries REMOVE, which removes its forwarding state, if the number of destination addresses in each Xcast packet sent from the upstream state node *u* does not exceed δ. Otherwise, state node *d* tries MOVE, which moves its forwarding state to the parent node, if the parent node of *d* is stateless and the number downstream state nodes is no more than δ.

The appendix in this paper proves that the above two simple operations lead to the optimal solution, i.e., minimum number of state nodes in each multicast tree, even though the sequence of REMOVE and MOVE operations generated by all state nodes is *arbitrary*. In other words, the distributed operations of each state node lead to the optimal assignment of forwarding states. In this paper, we present the protocol design for the two operations. In addition, we prove that the number of operations is limited, i.e., $O(n\delta k)$, in Section III, where *n* is the number of routers in the tree, and *k* is the maximum number of routers between two adjacent branching routers, i.e., routers with at least two child routers. In addition, we further propose an extension for MOVE to reduce the number of operations to $O(n\delta)$ in Section IV.

Xcast is a special case of our approach with δ as ∞. In this case, each packet can contain arbitrary number of destination addresses, and we thereby can directly include the addresses of all leaf nodes in each Xcast packet. In contrast, when δ equals one, only the branching nodes need to store the forwarding state, because any non-branching node has only one downstream state node and thereby is able to remove its forwarding state. Therefore, previous works [10][11] that maintain forwarding states in only the branching nodes are special cases of our approach.

B. *Protocol States and Messages*

Our protocol supports dynamic group membership and the rerouting of multicast trees when the network topology is changed. In addition, our protocol automatically removes, moves, or creates the state nodes in order to converge toward the new optimal assignment when the group membership or network topology is changed. Our protocol is loop-free as long as the underlying unicast routing is loop-free. Moreover, Section IV shows that we can incrementally deploy our protocol to gradually increase the scalability of a network that adopts existing multicast routing protocols.

The forwarding state of a multicast group stored in a router contains the following information.

- Group ID, which includes a class-D IP address, or both a class-D address and the address of the sender in SSM.
- Maximum number of addresses that can be included in each Xcast packet, i.e., δ.
- A Join timer
- A Move_Up timer
- Addresses of the parent node and the upstream state node

For each downstream interface, the forwarding state includes the following information.

- Addresses of the downstream state nodes. Each downstream state node is associated with a Leave timer.
- A Move_Down timer

Each leaf node of the tree only maintains a Join timer and stores the address of its upstream state node.

Our protocol contains the following five messages: Join, Leave, Inform_Up_Node, Move_Up, and Move_Down. In the following, we first briefly introduce each message. A Join message is always sent toward the root of the tree and intercepted by the first state node that receives it. A new leaf node connects to the existing multicast tree with a Join message, and the existing state node maintains a soft state with the message. When the Join timer expires, each state node sends a Join message to refresh the Leave timer in its upstream state node. If the network topology is changed, the message finds a new path to the new upstream state node. The second protocol message, Leave message, is always sent to the upstream state node. A state node removes the address of a downstream state node when the Leave message arrives or the corresponding Leave timer timeouts.

An Inform_Up_Node is always sent to a downstream state node to inform the state node of its new upstream state. A Move_Up message is always sent toward the upstream state node and intercepted by the neighbor node, which acts as the parent node. A state node moves its forwarding state to the parent node when all of the following conditions hold. 1) The Move_Up timer expires. 2) The parent node is not the upstream state node. 3) The number of downstream state nodes is no more than δ. The last message, Move_Down, handles the situation that a state node has more than δ downstream state nodes from a downstream interface due to the fact that a new leaf node joins or the network topology is changed. In this case, the downstream branching node has to store the forwarding state in order to keep the assignment of state nodes feasible. The state node uses a Move_Down message to move the forwarding state of the downstream interface to the closest downstream branching node. The Move_Down message is delivered via Xcast to find the closest downstream nodes. The addresses of all downstream state nodes from the overloaded downstream interface is included in the Xcast packet, and the first router that needs to duplicate the packet to relay to more than one neighbor node is the closest downstream branching node.

In order to prevent data loss, when a state node changes its upstream state node, it sends a Leave message to its original upstream state node only after an Inform_Up_Node arrives from the new upstream state node. The Inform_Up_Node message is a guarantee that the new upstream state node will deliver the data to the state node. Similarly, a state node can remove its forwarding state only after all downstream state nodes change their upstream state nodes.

In the following, we describe the behavior of each state node *n* after *n* receives a protocol message sent from another state node *m* in great detail. The information contained in each message is located within the parenthesis, where *r* is the root of the tree, u_m is the upstream state node of *m*, D_m is the set of downstream state nodes of *m*. In addition, $D_{n,m}$ is the set of downstream state nodes of *n* from the downstream interface to *m*. In other words, the addresses of *m* and all the other state nodes in $D_{n,m}$ are located in the same Xcast packet sent from the downstream interface of *n* to *m*.

1) ***Join***(u_m, D_m)*:* If $u_m \neq n$ or $m \notin D_{n,m}$, node *n* adds *m* to $D_{n,m}$ and D_n, and sends Inform_Up_Node() to *m* to notify that *n* is the new upstream state node of *m*. Node *n* also starts the Move_Down timer if $|D_n| > \delta$ holds. Otherwise, node *n* considers whether the forwarding state of *m* can be removed. If $|D_m \cup D_{n,m}| - 1 \leq \delta$ holds, node *n* adds D_m to $D_{n,m}$ and D_n, and sends Inform_Up_Node() to every *q*, to remove the forwarding state of node *m*, $\forall q \in D_m$.

2) ***Leave****():* Node *n* removes *m* from $D_{n,m}$ and D_n. If D_n is empty, node *n* removes the forwarding state and sends Leave() to u_n.

3) ***Inform_Up_Node****():* If *m* is not u_n, node *n* sends Leave() to u_n and sets u_n as *m*.

4) ***Move_Up***(D_m)*:* Node *n* first sends Join($r, \varnothing$) toward the sender to find u_n. Then, if Inform_Up_Node() arrives from a state node *p*, node *n* creates the forwarding state, sets u_n as *p*, $D_{n,m}$ and D_n as D_m, and sends Inform_Up_Node() to every *q* to remove the forwarding state of node *m*, $\forall q \in D_m$.

5) ***Move_Down***(D_m)*:* In this message, *n* is the first downstream branching node that receives the message. After receiving the message, node *n* sends Join($r, \varnothing$) toward the sender of the group. Then, if Inform_Up_Node() arrives from a state node *p*, node *n* creates the forwarding state and sets u_n as *p*. Moreover, for every *q* in D_m, node *n* finds the outgoing downstream interface to *q*, adds *q* to D_n and $D_{n,q}$, and sends Inform_Up_Node() to *q*.

C. Protocol Operations

In the following, we explain the protocol operations with an example in Fig. 1.

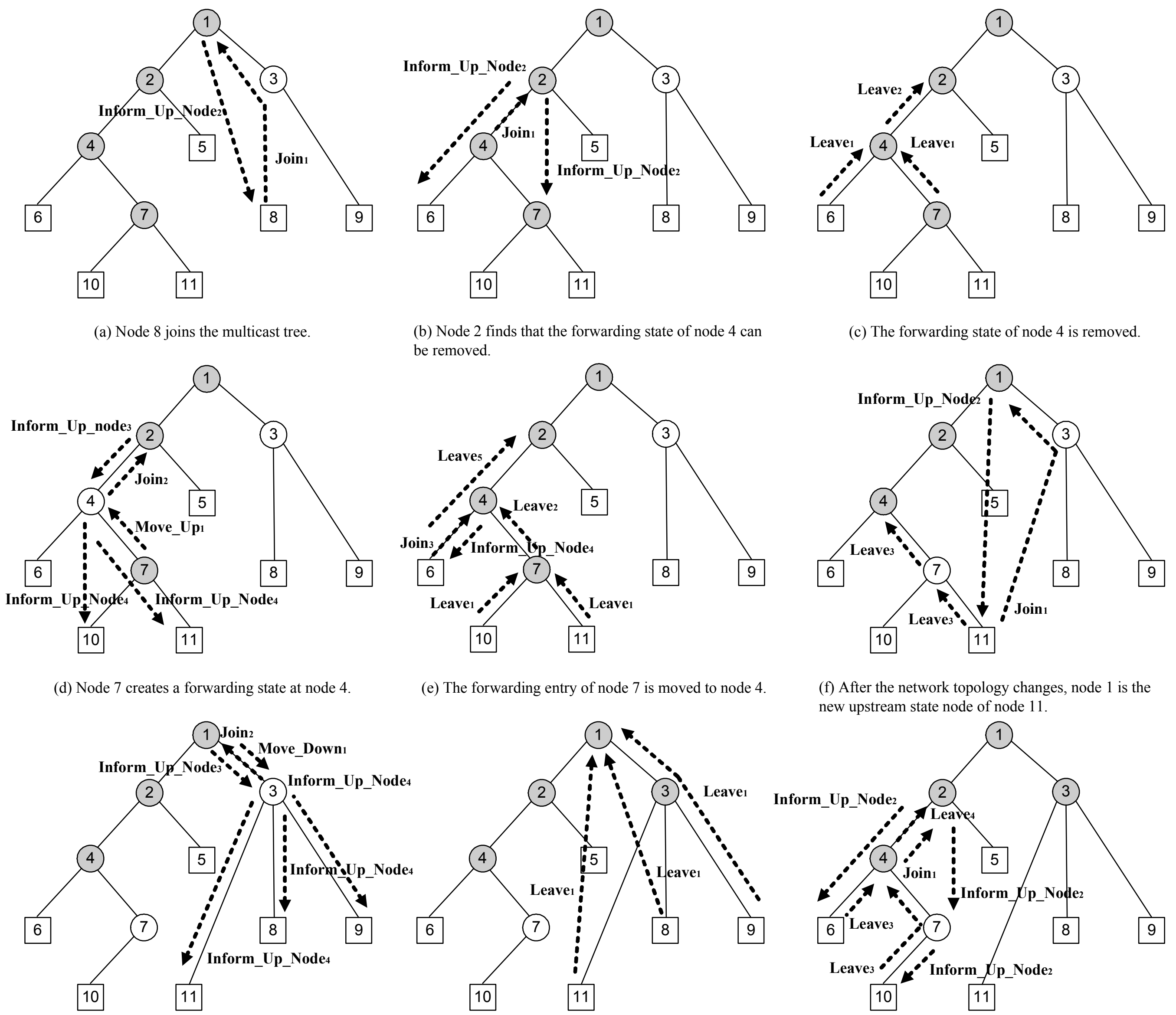
1) Joining the multicast tree: When a new receiver decides to join a group, it sends a Join message along the shortest path toward the root of the tree to connect to the existing multicast tree. The first state node that receives the message stops forwarding the message and acts as the upstream state node of the new receiver. It returns an Inform_Up_Node message to the new receiver to notify its identity. In Fig. 1 (a), node 8 decides to join the multicast group and sends a Join message toward the root. Node 1 receives the message, adds node 8 to its forwarding state, and replies with an Inform_Up_Node message. From this message, node 8 understands that node 1 is its upstream state node.

2) Leaving the multicast tree: Each receiver sends a Leave message to its upstream state node when it decides to leave the group. A state node sends a Leave message to its upstream state node when it changes its upstream state node or when there is no downstream state node in the forwarding state.

3) Removing the forwarding state: After the upstream state node receives a Join message from a state node, it determines if the state node, which is denoted as a *removed state node* here, can remove its forwarding state. The Join message contains the addresses of all downstream state nodes of the removed state node. Therefore, the upstream state node can find the number of downstream state nodes from the interface to the removed state node, assuming that the removed state node becomes stateless. If the removed state node can become stateless, the upstream state node adds all downstream state nodes of the removed state node to its forwarding state, and sends Inform_Up_Node messages to them to initiate a REMOVE operation. Each downstream state node changes its upstream state node in the forwarding state and sends a Leave message to its original upstream state node, i.e. the removed state node. After the removed state node receives the Leave messages from all downstream state nodes, it removes the forwarding state and sends a Leave message to its upstream state node.

In Fig. 1 (b), after node 2 receives the Join message from node 4, it finds that the forwarding state of node 4 can be removed since node 4 has only two downstream state nodes. Therefore, node 2 adds nodes 6 and 7 to its forwarding state and sends Inform_Up_Node to nodes 6 and 7. In Fig. 1 (c), nodes 6 and 7 regard node 2 as the new upstream state node and send Leave messages to their original upstream node, which is node 4. Since node 4 has no downstream state node in its forwarding state, it removes its forwarding state and sends a Leave message to its upstream state node, namely, node 2.

4) Moving the forwarding state to the parent node: If the total number of downstream state nodes is no larger than δ, a state node, which is called a *moved state node* here, can move its forwarding state to its parent node. When the Move_Up timer expires, the moved state node sends a Move_Up message to its parent node. The parent node prepares to act as a state node and finds its upstream state node by sending a Join message upstream, and it creates the forwarding state after an Inform_Up_Node arrives. Then, the parent node includes all downstream state nodes of the moved state node in its forwarding state, and sends Inform_Up_Node messages to them. The addresses of these nodes are carried in the Move_Up message. Each downstream state node changes its upstream state node in the forwarding state and sends a Leave message to the original upstream state node, i.e. the moved state node. After the moved state node receives the Leave messages from all downstream state nodes, it removes the forwarding state and sends a Leave message to its upstream state node if it is not a receiver.



(a) Node 8 joins the multicast tree.

(b) Node 2 finds that the forwarding state of node 4 can be removed.

(c) The forwarding state of node 4 is removed.

(d) Node 7 creates a forwarding state at node 4.

(e) The forwarding entry of node 7 is moved to node 4.

(f) After the network topology changes, node 1 is the new upstream state node of node 11.

(g) Since node 1 has three downstream state nodes from the interface to node 3, it creates the forwarding state at node 3.

(h) The forwarding state of node 1 from the interface to node 3 is moved to node 3.

(i) After the forwarding state of node 4 is removed, the assignment of the state nodes is optimal.

Fig. 1. An example of protocol operations with $\delta = 2$. The sequence of the messages in the example is identified by the subscript following each message.

In Fig. 1 (d), node 7 decides to move its forwarding state to node 4. It sends a Move_Up message to node 4, and node 4 sends a Join message toward the root to find the upstream state node. After node 4 receives an Inform_Up_Node from node 2, it creates the forwarding state, adds nodes 10 and 11 to its forwarding state, and sends Inform_Up_Node to nodes 10 and 11. In Fig. 1 (e), nodes 10 and 11 know that node 4 is the new upstream state node and send the Leave messages to the original upstream state node, which is node 7. Since node 7 has no downstream state node in its forwarding state, it removes its forwarding state and sends a Leave message to its upstream state node, which is node 4. Since node 4 is a new state node, node 6 has to update its upstream state node in its forwarding state. When node 6 sends a Join message toward the sender, the message is intercepted by node 4. Since node 6 is not a downstream state node in the forwarding state, node 4 adds node 6 to its forwarding state and returns an Inform_Up_Node message. Then node 6 sends a Leave message to its original upstream state node, which is node 2.

5) Rerouting the multicast tree: When a link of the path connecting two state nodes breaks, the underlying unicast routing protocol will try to reroute the path. In this case, the recovering path is probably longer than the original one. Therefore, in our protocol, a state node periodically checks if there is a new path connecting to a new upstream state node by sending a Join message toward the sender. When a Join message arrives at a state node that is not the original upstream state node, the state node becomes the new upstream state node and returns an Inform_Up_Node message. Then, the state node that sends the Join message changes its upstream state node in the forwarding state and sends a Leave message to its original upstream state node. In Fig. 1 (f), we assume the network topology changes, and the Join message passes through node 3 to node 1, instead of to its original upstream state node, which is node 4. Node 1 returns an Inform_Up_Node message to node 11. Node 11 updates its upstream state node in its forwarding state and sends a Leave message to its original upstream state node, i.e., node 4.

6) Moving the forwarding state to the closest downstream branching node: A downstream interface is overloaded if the number of downstream state nodes from the interface is more than δ. This situation occurs when the network topology changes or when a new receiver joins. In this case, the state node, which is called a moved state node here, needs to create the forwarding state in the closest downstream branching node. Therefore, the moved state node now has only one downstream state node from the overloaded interface, and the original downstream state nodes become the downstream state nodes of the downstream branching node. The corresponding protocol operation is described as follows. The moved state node initiates the Move_Down timer of a downstream interface when the interface is overloaded. We use the timer to address the case that an interface is temporarily overloaded during a remove or move operation. When the timer expires, the moved state node sends a Move_Down message via Xcast to all downstream state nodes from the overloaded interface. The first stateless node that needs to forward the message from more than one outgoing interface is the closest downstream branching node. The node intercepts the message and sends a Join message upstream to inform its identity. After the branching node receives an Inform_Up_Node, it sends Inform_Up_Node to the downstream state nodes, where the addresses of these nodes are the destination addresses of the Move_Down message. Finally, each downstream state node understands that it can receive data from a new upstream state node and sends a Leave message to its original upstream state node, i.e. the moved state node.

In Fig. 1 (g), since the downstream interface to node 3 is overloaded, node 1 moves the forwarding state of the interface to its downstream branching node. It sends a Move_Down message downstream to nodes 8, 9, and 11, and the message is intercepted by node 3 since node 3 has three outgoing interfaces. Node 3 returns a Join message to notify its identity. After node 3 receives an Inform_Up_Node message from node 1, it adds nodes 8, 9, and 11 to its forwarding state and sends Inform_Up_Node to them. Then, nodes 8, 9, and 11 in Fig. 1 (h) then send Leave messages to their original upstream state node, which is node 1. Finally, after node 4 removes its forwarding state in Fig. 1 (i), the state assignment is optimal.

III. Overhead Analysis

The number of protocol messages and the convergence time are proportional to the total number of REMOVE and MOVE operations required to obtain the optimal assignment. We prove that the number of REMOVE and MOVE is $O(n\delta k)$ as follows, where n is the number of routers in the tree, and k is the maximum number of routers between two adjacent branching routers. Initially, we assume that every router in the tree stores the forwarding states. A state can be removed at most once but can be moved upstream many times. However, a state can be moved upstream and traversed at most $\delta - 1$ branching nodes, because the number of downstream states nodes in the state increases at least by one when the state arrives at a branching node, due to at least one downstream state node from other downstream interfaces of the branching node. The number of downstream states nodes cannot exceed δ; otherwise, the forwarding state cannot be moved upstream to a non-branching node. Therefore, a state can be moved upstream $O(n\delta k)$ times because the number of MOVE operations between two adjacent branching nodes is $O(k)$.

In the next section, we show that the number of operations can be further reduced to $O(n\delta)$ with an improved MOVE operation. The number of protocol messages is $O(n\delta^2)$ in this case because each operation leads to $O(\delta)$ Inform_Up_Node and Leave messages. In addition, we believe that the convergence time can be much smaller than shown above since multiple nodes can perform the two operations simultaneously in a distributed manner.

IV. Discussion

1) Load balancing of forwarding states: In previous sections, we focus on minimizing the number of state nodes in each multicast tree. We extend our protocol to avoid state overload in some routers as follows. When a router temporarily with many states receives a Move_Up message of a group, it ignores the message or forwards the message to the next-hop node toward the root of the tree, to avoid storing the forwarding state of the group. In case that a router is overloaded, we move the states of some groups to the corresponding downstream branching routers from every downstream interface, with the operation described in Section III, to reduce the number of states stored in the router. Therefore, for the backbone routers with large degrees [16], which tend to be branching routers in a tree, our protocol can regard the routers as overloaded routers, and this approach enables the backbone routers to store no forwarding state.

2) Moving the forwarding state to the closest upstream branching node: In operation MOVE, a router moves the forwarding state to the parent node. If the parent node is not a branching node, the state can be further moved up because the number of downstream state nodes of these non-branching nodes is the same. Therefore, we can improve MOVE by directly moving the forwarding state to the closest upstream branching node. To find the closest upstream branching node, the state node first sends a new message, Query_Branch_Node, to the upstream state node. The upstream state node then sends another new message, Find_Branch_Node, back to the state node, with the addresses of the upstream state node's downstream state nodes included in the message. Every node traversed by the message can find whether it is a branching node with these addresses. The message then records the address when traversing a branching node, and the last address thereby corresponds to the closest upstream branching nodes.

Afterwards, the state node sends Move_Up to the upstream branching node to move the forwarding states. In this modified MOVE, a state can be moved directly between branching nodes, thereby reducing the number of MOVE operations to $O(n\delta)$.

3) Robustness and convergence: Our protocol is based on the binding of the upstream state node and the downstream state node. The binding is maintained by the periodical Join message from the downstream state node and the Inform_Up_Node message from the upstream state node. Therefore, when any node *n* fails, the above two messages are not delivered by *n*, and our protocol in this case breaks the binding between the failed node *n* and any other node. In addition, for each downstream state node *m* of *n*, the underlying unicast routing protocol finds the new path for the Join message periodically sent by *m*, and the message initiates a new binding between *m* and the new upstream state node of *m*. Therefore, our protocol is able to handle the failure of the network and the change of the unicast routing.

4) Incremental deployment based on existing multicast routing protocols: A router that does not support our protocol cannot act as a state node and will not intercept the Join and Move_Up messages. The destinations of our five protocol messages are state nodes in the existing multicast tree, and our messages can therefore bypass the routers that do not support our protocol. In addition, our protocol can be incrementally deployed on the unidirectional multicast trees conforming to the existing standards, such as PIM-SM, to gradually increase the scalability of the network. Current PIM-SM supports multi-access transit LANs, which represents an interface that can connect multiple adjacent routers. Our idea is to add a virtual multi-access interface to each physical interface of a router that supports PIM-SM and our protocol. The routers in the virtual multi-access network are the downstream state nodes of our protocol, and the router sends packets via Xcast to the virtual multi-access network. Therefore, the number of routers in the virtual multi-access network is bounded by δ. After the router finds its upstream state nodes, it leaves the existing PIM-SM multicast tree with the Prune message in PIM-SM. Therefore, the routers between the state node and its downstream state nodes can remove the PIM-SM forwarding states, thereby incrementally improving the scalability of the network.

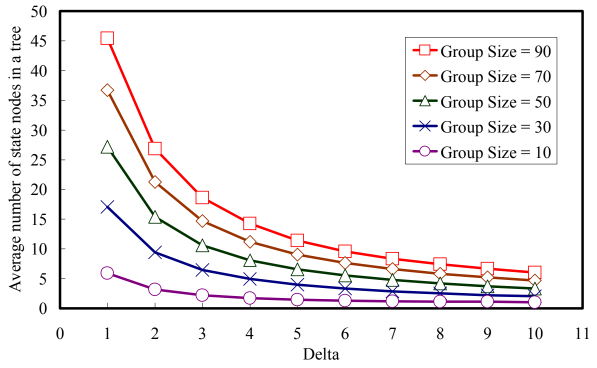
V. Simulation Results

Our simulation results show that the above two problems are solved by the proposed algorithms. We use small flat graphs with Waxman distribution [12] as the network topologies to evaluate the performance of our algorithms in networks with different graph characteristics. We also use MBone [13], a real multicast tree traced by Mwalk [14][15], and large graphs with the power-law distribution generated by Inet [16][17] to test our algorithms in more realistic networks. The simulation results are averaged over 100 samples. The input parameters are summarized as follows.

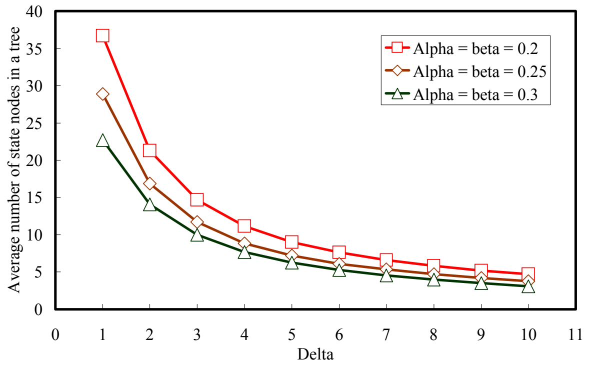
- Graph characteristic. We use the following settings as the parameters of the Waxman distribution in the flat graphs: $(\alpha = 0.2, \beta = 0.2)$, $(\alpha = 0.25, \beta = 0.25)$, and $(\alpha = 0.3, \beta = 0.3)$. The graphs with larger values of α and β have larger node degrees and smaller graph diameters [18]. Therefore, the height of a multicast tree in the graphs is smaller, and a node in a multicast tree has more child nodes.
- Group size. The group size is the number of receivers in a multicast tree.
- The maximal number of destinations from a downstream interface, δ. When δ equals one, all branching nodes are the only state nodes in a multicast tree.
- Distribution of receivers. We use the affinity and disaffinity model [19] to describe the distribution of receivers in a multicast tree. The distribution of receivers is correlated with the topology of a multicast tree. With a positive affinity index, receivers tend to cluster together, and the distribution is suitable for applications such as local news and traffic reports. With a negative affinity index, receivers tend to spread out, and the distribution is proper for applications such as video conferencing. When the affinity index is zero, all receivers are chosen uniformly at random among all nodes.

Figs. 2, 3, and 4 compare the results with different parameters in networks with the Waxman distribution. The network has 100 nodes. Fig. 2 compares the results with respect to different values of δ, different group sizes, and different values of α and β in the Waxman distribution. We measure the average number of state nodes in a multicast tree. In Fig. 2 (a), a multicast tree with a larger group size has more state nodes. The number of state nodes in a multicast tree decreases as δ increases. Moreover, a small δ is sufficient to eliminate over 50% forwarding states. Therefore, our mechanism can deliver data with only a few forwarding states without incurring a large forwarding delay. In Fig. 2 (b), a graph with larger α and β uses fewer state nodes in a tree. In this case, the depth of a multicast tree decreases, and each node in a tree has more child nodes. Therefore, each state node can serve more receivers and downstream state nodes, so fewer nodes have to store the forwarding states. Moreover, the results of different α and β converge as δ increases because fewer nodes need to store the forwarding states.

Fig. 3 compares the results with different group sizes, different α and β in the Waxman distribution, and different affinity indices. We measure the percentage of nodes with the forwarding states in a multicast tree. In Fig. 3 (a), more nodes in a multicast tree store the forwarding states as the group size increases. For a graph with larger α and β, receivers tend to connect to on-tree nodes with larger degrees. Therefore, a multicast tree requires only slightly more nodes to store the forwarding states as the group size increases. In Fig. 3 (b), receivers tend to cluster together as the affinity index increases. Fewer nodes in a multicast tree store forwarding states because a node outside a cluster may need to store only one state for the whole cluster. The results of different affinity indices converge

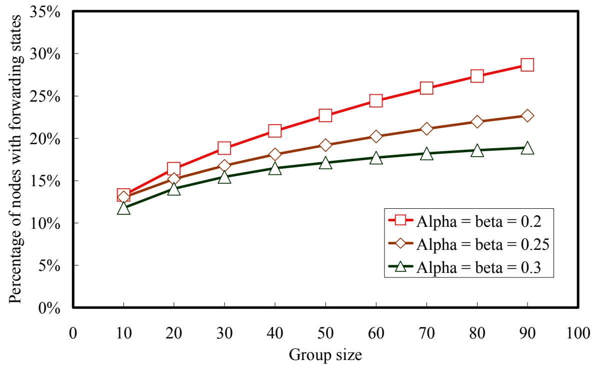


(a) $\alpha = \beta = 0.2$

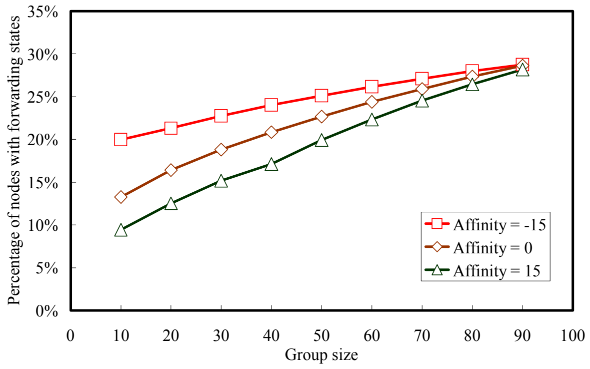


(b) group size = 70

Fig. 2. Average number of state nodes in a multicast tree with different δ, different group sizes, and different values of α and β in the Waxman distribution, affinity index = 0.

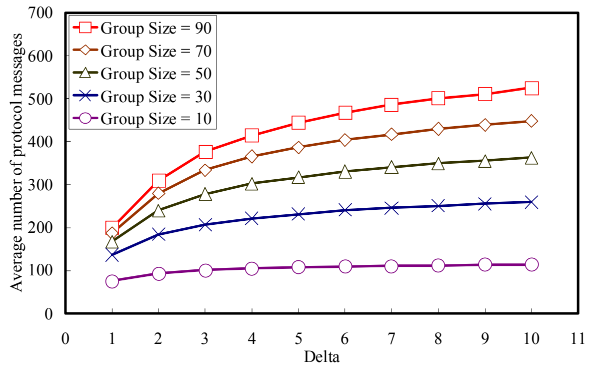


(a) affinity index = 0

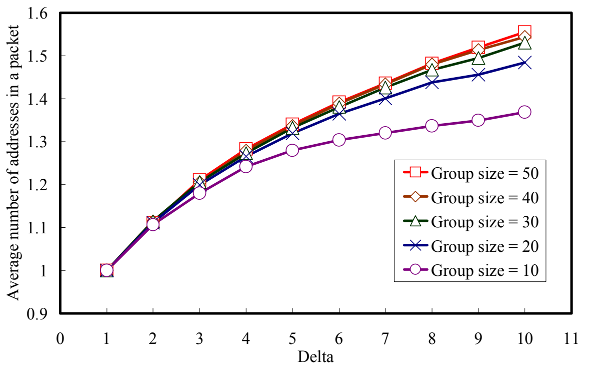


(b) $\alpha = \beta = 0.2$

Fig. 3. Percentage of nodes with the forwarding states in a multicast tree with different group sizes, different values of α and β in the Waxman distribution, and different affinity indices, $\delta = 2$.



(a)



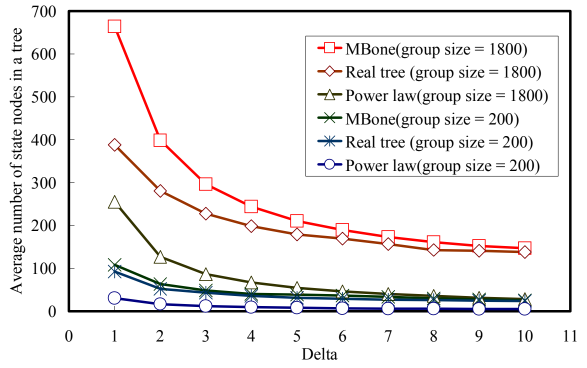
(b)

Fig. 4. Protocol overheads with different δ and different group sizes, $\alpha = \beta = 0.2$, affinity index = 0.

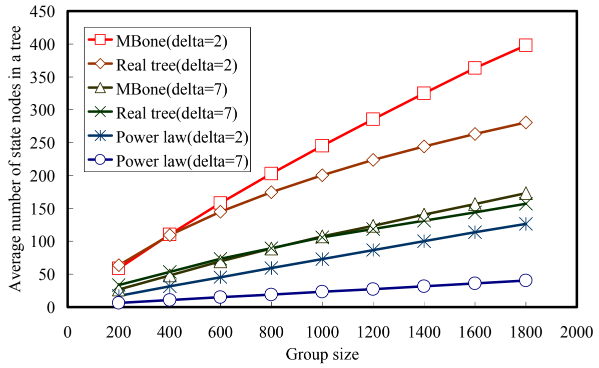
as the group size approaches 100 because the distribution of receivers becomes similar.

We measure the protocol overhead in Fig. 4. Our protocol generates more messages as the group number increases in Fig. 4 (a). However, the number of protocol messages converges as δ increases because we have more chances to remove a forwarding state of a router directly, instead of first moving the forwarding state to the parent node. Fig. 4 (b) shows the average number of addresses in each packet is much smaller than δ because the number of addresses in a packet dramatically decreases when the packet arrives at a branching router. Moreover, each leaf node of a tree receives a packet with only one address. A packet contains more addresses when a group has more receivers. However, it converges as the group size increases because a larger group requires more state nodes.

We next measure the average number of state nodes in a multicast tree for *MBone*, a real multicast tree, and large networks with the power-law distribution, as shown in Fig. 5. The number of state nodes in a multicast tree decreases as δ increases. A small value of δ is sufficient to eliminate most of the forwarding states. A multicast group with larger group sizes has more state nodes. A tree in power-law graphs uses much fewer state nodes than the others because a power-law graph has a few nodes with very large degrees. Therefore, the diameter of a power-law graph is smaller than that in *MBone*,

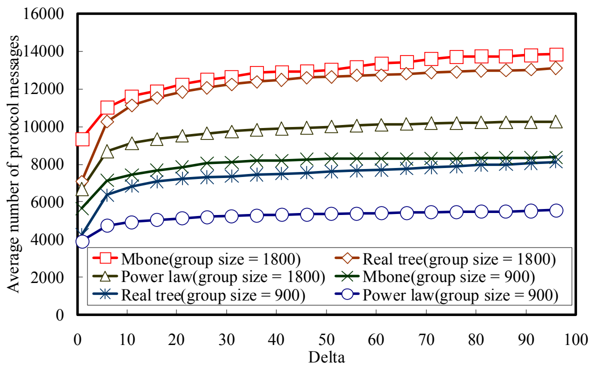


(a)

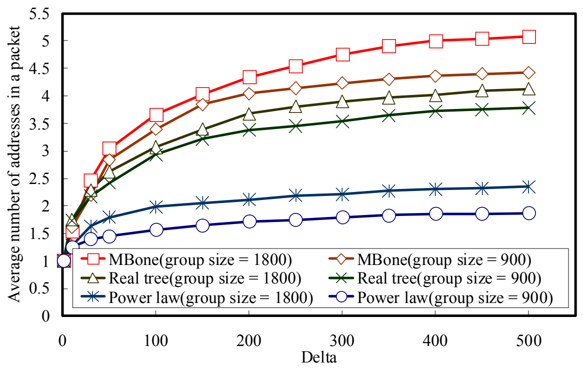


(b)

Fig. 5. Average number of state nodes in a multicast tree in different graphs with different δ and different group sizes.



(a)



(b)

Fig. 6. Protocol overheads in different networks with different δ and different group sizes.

and the height of a multicast tree in a power-law graph is smaller than the height of a tree in the other two graphs.

Fig. 6 shows the protocol overhead. The number of protocol messages converges as δ increases in Fig. 6 (a). In Fig. 6 (b), the average number of addresses in a packet in power-law networks is much smaller than that in *MBon*e and in the real multicast tree, because power-law networks have a few nodes with very large degrees. These nodes tend to have much more child nodes, and there are fewer receivers or downstream branching routers from each downstream interface. Therefore, each packet sent from a downstream interface has fewer destination addresses.

VI. Conclusion

In this paper, we propose a scalable and adaptive protocol for multicast communications. Our protocol is scalable in terms of both the group number and group size. We avoid the disadvantages of IP multicast and Xcast by choosing only a few routers to store the forwarding states, and multicast packets are delivered via Xcast among theses routers. Our protocol is simple and can obtain the optimal assignment of forwarding states among routers. The assignment of forwarding states in our protocol is adaptive to the dynamic group membership and the change of network topology. In addition, we prove that the overhead of our protocol is limited. Moreover, our protocol can be deployed incrementally and cooperate with the existing IP multicast routing protocols to gradually increase the scalability of the network.

Acknowledgement

This work was supported in part by the Excellent Research Projects of National Taiwan University, under Grant Number 97R0062-06, and in part by National Science Council (NSC), Taiwan, under Grant Number NSC96-2628-E-002-003-MY3.

References

[1] B. Cain, S. Deering, I. Kouvelas, B. Fenner, and A. Thyagarajan, "Internet group management protocol, version 3," *IETF RFC 3376*, Oct. 2002.

[2] D. Waitzman, C. Partridge, and S. Deering, "Distance vector multicast routing protocol," *IETF RFC 1075*, Nov. 1988.

[3] J. Moy, "Multicast routing extensions for OSPF," *Communications of the ACM*, vol. 37, no. 8, pp. 61–66, Aug. 1994.

[4] T. Ballardie, P. Francis, and J. Crowcroft, "Core based trees (CBT)," *ACM SIGCOMM*, pp. 85–95, 1993.

[5] B. Fenner, M. Handley, H. Holbrook, I. Kouvelas, "Protocol independent multicast - sparse mode (PIM-SM): protocol specification (Revised)," *IETF RFC 4601*, Aug. 2006.

[6] D. Thaler, M. Handley, and D. Estrin, "The Internet multicast address allocation architecture," *IETF RFC 2908*, Sep. 2000.

[7] H. Holbrook and B. Cain, "Source-specific multicast for IP," *IETF RFC 4607*, Aug. 2006.

[8] T. Wong and R. Katz, "An analysis of multicast forwarding state scalability," *IEEE ICNP*, pp. 105–115, 2000.

[9] R. Boivie, N. Feldman, Y. Imai, W. Livens, D. Ooms "Explicit multicast (Xcast) concepts and options," *IETF RFC 5058*, Nov. 2007.

[10] J. Tian and G. Neufeld, "Forwarding state reduction for sparse mode multicast communication," *IEEE INFOCOM*, pp. 711–719, 1998.

[11] I. Stoica, T. S. E. Ng, and H. Zhang, "REUNITE: a recursive unicast approach to multicast," *IEEE INFOCOM*, pp. 1644–1653, 2000.

[12] B. M. Waxman, “Routing of multipoint connections,” *IEEE Journal on Selected Areas in Communications*, vol. 6, no. 9, pp. 1617–1622, Dec. 1988.
[13] USC/ISI SCAN project, http://www.isi.edu/scan/mbone.html.
[14] R. Chalmers and K. Almeroth, “On the Topology of Multicast Trees”, *IEEE/ACM Transactions on Networking*, vol. 11, no. 1, pp. 153–165, Feb. 2003.
[15] Mwalk project, http://www.nmsl.cs.ucsb.edu/mwalk/.
[16] H. Tangmunarunkit, R. Govindan, and J. Jamin, “Network topology generators: degree-based vs. structurel,” *ACM SIGCOMM*, pp. 147–159, 2002.
[17] Inet topology generator, http://topology.eecs.umich.edu/inet/.
[18] E. W. Zegura, K. L. Calvert, and M. J. Donahoo, “A quantitative comparison of graph-based models for Internet topology,” *IEEE/ACM Transactions on Networking*, vol. 5, no. 6, pp. 770–783, Dec. 1997.
[19] G. Phillips, S. Shenker, and H. Tangmunarunkit, “Scaling of multicast trees: comments on the Chuang-Sirbu scaling law,” *ACM SIGCOMM*, pp. 41–51, 1999.

APPENDIX

The following theorem proves that operations REMOVE and MOVE find the optimal assignment of multicast forwarding states.

Theorem 1. *Given an optimal assignment S^*, for any feasible assignment S that is not optimal, we can obtain another optimal assignment by a sequence of operations on S^* and S.*

Proof: We prove this theorem by introducing an algorithm with a sequence of operations on S^* and *S*. After the algorithm stops, S^* and *S* are identical, and both are optimal. For each node *m* in tree *t*, let σ_m^* and σ_m denote whether node *m* in S^* and S, respectively, is a state node. The algorithm is specified as follows.

```
1.  Number all nodes from 1 to $|V_t|$ with postorder traversal.
2.  For m from $|V_t|$ to 2, $m \notin R_t$,
      if $\sigma_m^* = 0$ and $\sigma_m = 1$, then let $\sigma_m \leftarrow 0$ and $\sigma_{p_m^t} \leftarrow 1$;
      else if $\sigma_m^* = 1$ and $\sigma_m = 0$, then let $\sigma_m^* \leftarrow 0$ and
          $\sigma_{p_m^t}^* \leftarrow 1$;
      end if;
    end for.
```

Step 2 of the above algorithm compares node *m* in *S* and S^*. If *m* is a state node in *S* only, node *m* removes its forwarding state when its parent node is a state node. However, node *m* moves the forwarding state to its parent node when its parent node is stateless. At the beginning of the iteration, for all node *m*, we maintain both assignments as feasible solutions to the problem, and both assignments of state nodes in the sub-tree rooted at *m* must be identical, except node *m*, because all nodes except *m* in the sub-tree have been compared. At the end of each iteration, the upstream state node of *m* in S^* is either identical or upstream to the upstream state node of *m* in *S*. Therefore, *S* is a feasible assignment since S^* is also feasible. In other words, both assignments are feasible at the end of the iteration. Since both assignments are feasible before step 1, the above induction proves that the above algorithm maintains feasibility of both assignments at the end of each iteration. Moreover, the assignment of state nodes in *S* is identical to S^* after the algorithm stops. Since the number of state nodes in S^* is not changed, S^* is still an optimal assignment. Therefore, *S* is also an optimal assignment after the algorithm stops.

Therefore, an assignment is optimal if no state node can remove its forwarding state or move its forwarding state to its parent node. Otherwise, if the assignment is not optimal. we can obtain an optimal assignment after a sequence of operations according to Theorem 1, and the above statement contradicts the condition that no state node can remove or move its forwarding state.