A Tree Switching Protocol for Multicast State Reduction[1]

Filli Y.Y. Cheng Rocky K.C. Chang
The Hong Kong Polytechnic University
Hung Hom, Kowloon, Hong Kong
E-mail: {csycheng, csrchang}@comp.polyu.edu.hk

Abstract

In this paper, we propose a new Tree Switching Protocol (TSP) to reduce multicast routing states required for a forest of multicast trees. The protocol accomplishes this goal by selecting a base multicast tree and "switching" other multicast trees to the base tree. This results in more overlapping among the multicast trees and a net reduction in multicast state. A further reduction in the state is possible by applying state aggregation to the overlapped tree branches. Simulation results show that the tree switching operation alone could result in a very significant state reduction. The TSP is a distributed protocol running on top of any protocol independent multicast routing protocols. It is also loop-free and very efficient.

1. Introduction

Multicast routing from a source to a group of members generally involves construction of multicast trees. Multicast tree can be classified into source-specific trees (e.g. DVMRP, MOSPF, PIM-DM), shared trees (e.g. CBT) and a mixture of the two (e.g. PIM-SM [1], QoSMIC).

One important criterion for evaluating a particular multicast routing protocol is its scalability. Since the source-specific approach requires a separate tree for each active source, it is not scalable. The shared tree approach is more scalable since only one tree is required for any number of sources. However, the scalability of this approach degrades as the number of multicast groups increases. To make the multicast routing scalable to both number of sources and number of multicast groups, several hierarchical approaches (e.g. HDVMRP, HPIM, OCBT, BGMP) were proposed. Recently, a number of proposals were made to aggregate multicast state in order to reduce the multicast state storage [2-5].

In terms of terminology, a *router* in this paper is referred to a multicast router, which belongs to at least one multicast tree. A *designated router* is a router that is connected directly to at least a member host in a multicast group or to a multicast source. An *on-tree router* is one that is not connected directly to member hosts but it simply forwards multicast packets to downstream nodes. A router is either a *leaf router* or an *interior router* on a multicast tree. A leaf router terminates the forwarding of multicast packets and others are interior routers. A leaf router must be a designated router.

In this paper we propose a novel approach to reduce the multicast state requirement. This approach is based on the following observation. The number of multicast trees (whether shared trees or source-specific trees) existing concurrently within a domain or in the Internet backbone could be very large as multicast-based applications become more popular. Consequently, it is likely that two or more multicast trees could partially overlap with each other. In other words, a multicast router may belong to several multicast trees. Based on this observation, we propose a tree switching operation to reduce the total multicast state requirement for a forest of multicast trees. The idea is that a designated router could select a multicast tree out of all the trees that it belongs as a base multicast tree. It will attempt to switch other trees to the base tree. If successful, all other tree branches, except for the base tree, can be pruned away. If the number of routing entries pruned away is higher than the number of new entries added to the base tree, the tree switching will result in a net reduction in the amount of states. As we shall see from the simulation results later, the tree switching operation could indeed reduce the multicast state significantly. Moreover, the tree switching also increases the overlapping among multicast trees. A further state reduction can be accomplished by applying state aggregation to the overlapped tree branches.

The main contribution of this paper is to propose a new *Tree Switching Protocol* (TSP) for performing the

[1] This work was partially supported by the Hong Kong Polytechnic University Central Research Grant V632.

0-7695-0722-0/00 $10.00 © 2000 IEEE

tree switching operation. TSP is a distributed protocol, routing loop free, and very efficient. This approach is most suitable for both intra-domain and inter-domain levels because those multicast sessions tend to be long-lived and scalability is a prime concern.

2. Overview of tree switching protocol

To give an overview on how TSP operates, we use PIM-SM as the underlying multicast routing protocol. That is, we assume that there are already a number of shared trees, each of them has a *rendezvous point* (RP) as the tree root and is identified by a unique class D address. We employ the following notations for the rest of this paper.

1. T, G: The set of shared trees and the corresponding set of multicast addresses under consideration.
2. T_i, g_i: The ith shared tree with multicast address g_i.
3. $(*, g_i, I_i, O_i)$: A multicast routing entry for multicast group g_i. The first parameter indicates the source host's address. This parameter is a wildcard for shared trees. I_i is an incoming interface, and O_i is an outgoing interface list for the multicast address g_i.

2.1. An example

The key ideas behind TSP can be best explained by the example in Figure 1. There are three multicast groups $g_{1\text{-}3}$ and we show parts of $T_{1\text{-}3}$ in the figure. In Figure 1(a), R_1 is a designated router for all three groups whereas the other three routers are on-tree routers. Moreover, R_1 is a leaf router on $T_{1\text{-}3}$. We first define the meaning of tree overlapping. Multicast trees T_j and T_k are said to be *overlapped on router* R_i if and only if $I_j = I_k$ and $O_j = O_k$ in the two corresponding routing entries in R_i; otherwise they are not overlapped on R_i. Therefore in this example only T_1 overlaps with T_3 on R_1 and R_2.

In this particular case, R_1 will attempt to switch $T_{2\text{-}3}$ to T_1. We call T_1 a *base multicast tree* and the other two *target multicast trees*. The whole process here involves three protocol messages: *SWITCH()*, *CONFIRM()* and *PRUNE()*. We will delay the detail protocol description to the next section. For the time being, it suffices to understand that the *SWITCH()* message is a request message for switching $T_{2\text{-}3}$ to T_1. Since T_1 overlaps with T_3 on R_1, tree switching is not necessary for T_3. Instead, in Figure 1(b), we show that the protocol triggers R_1 to combine both entries into a logical entry. All of the above take place as the step 1 in the figure.

The *SWITCH()* message also triggers R_2 to combine the entries for g_1 and g_3 since T_1 also overlaps with T_3 on R_2 (step 2). When the message arrives at R_3, R_3 responds with a *CONFIRM()* message back to R_1 and the message confirms that T_2 can be switched to T_1 since R_3 is also on T_2. At the same time, R_3 adds the interface 2 to the g_2's entry so that routers downstream could receive it (step 3). By receiving the *CONFIRM()* message, the interior router R_2 includes g_2 into its multicast routing entry since all three trees overlap with each other on this router (step 4).

When the message eventually arrives at R_1, it updates g_2's entry accordingly and sends a *PRUNE()* message to interface 9 to prune away T_2's branch as far as possible (step 5). R_4 is required to prune away g_2's entry upon receiving the *PRUNE()* message and possibly forwards the message to the upstream router (step 6). The tree switching operation is thus successful and the resulted routing entries are shown in Figure 1(c).

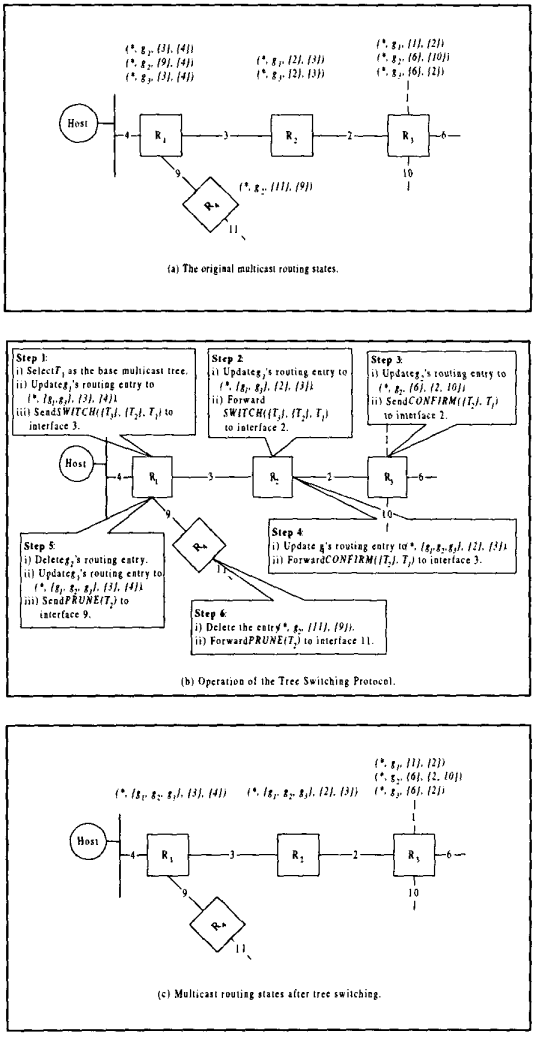

Figure 1. An example to illustrate tree switching operation.

3. Tree switching protocol in details

The TSP is a distributed protocol. Therefore several tree switching operations may be active at any time. In describing the protocol operation, we will focus on a designated router R_i that initiates the protocol. The TSP distinguishes whether a router is a designated router or an on-tree router. Only designated router could initiate the TSP.

3.1. Wake-up procedure

Every designated router is equipped with a wake-up timer and the timer expires after a random period of time. The timer expiration "wakes" up a designated router by making a state transition to the *Wake-up* state. We let $\boldsymbol{T}(i) \subseteq \boldsymbol{T}$ be the set of multicast trees that R_i is part of. The designated router will perform the following when it enters into the *Wake-up* state.

1. R_i first discovers the set of multicast trees that it is part of and for which it is a leaf router. We label this set by $\boldsymbol{T}_l(i) \subseteq \boldsymbol{T}(i)$. The wake-up is considered successful if R_i is on at least two trees and $\boldsymbol{T}_l(i)$ is not empty; otherwise it goes back to the *Idle* state.
2. R_i selects a base multicast tree from $\boldsymbol{T}(i)$ for tree switching and we denote this tree as $T_b(i) \in \boldsymbol{T}(i)$. We do not require R_i to be a leaf router on $T_b(i)$. However, in this paper we assume $T_b(i) \notin \boldsymbol{T}_l(i)$ except when all the trees in $\boldsymbol{T}(i)$ are also members of $\boldsymbol{T}_l(i)$, that is, $\boldsymbol{T}(i) = \boldsymbol{T}_l(i)$. By selecting the base tree in this way, we could switch more target trees to a base tree. If indeed $T_b(i) \in \boldsymbol{T}_l(i)$, we need to remove $T_b(i)$ from $\boldsymbol{T}_l(i)$ for the remaining tasks.
3. Based on $T_b(i)$ and $\boldsymbol{T}_l(i)$, R_i determines the set of trees in $\boldsymbol{T}_l(i)$ that overlaps with $T_b(i)$ and the set that does not. We use operators $\Pi()$ and $X()$ to determine the two sets. For a given set of trees $\boldsymbol{T}_S$ and a tree T, $\Pi_i(\boldsymbol{T}_S, T)$ $(X_i(\boldsymbol{T}_S, T))$ gives a set of trees that belong to $\boldsymbol{T}_S$ and are (not) overlapped with T on R_i. Therefore $\Pi_i(\boldsymbol{T}_l(i), T_b(i))$ gives the set of trees overlapped with $T_b(i)$ while $X_i(\boldsymbol{T}_l(i), T_b(i)) = \boldsymbol{T}_l(i) - \Pi_i(\boldsymbol{T}_l(i), T_b(i))$ gives the set of trees not overlapped with $T_b(i)$.
4. R_i then combines the routing entries for the trees in $\Pi_i(\boldsymbol{T}_l(i), T_b(i))$ with $T_b(i)$'s entry into a single logical entry since they are overlapped on R_i.
5. R_i finally sends a $SWITCH(\Pi_i(\boldsymbol{T}_l(i), T_b(i)), X_i(\boldsymbol{T}_l(i), T_b(i)), T_b(i))$ message upstream on $T_b(i)$ and starts a timer for receiving possible *CONFIRM()* messages for this request.

It is important to highlight the fact that a designated router will only consider the trees, on which it is a leaf router, as target multicast trees for tree switching. The reason is being that a successful tree switching operation will result in branch pruning. Since the protocol always starts from leaf routers, the pruning will not create routing loops or partition in a multicast tree.

3.2. Tree switching procedure

By sending the *SWITCH()* message upstream, R_i is moved from the *Wake-up* state to the *Wait* state. We now consider an upstream router R_j on $T_b(i)$. When R_j receives the *SWITCH()* message, R_j moves from the *Idle* state to the *Wait* state and performs the following.

1. R_j first discovers $\Pi_j(\boldsymbol{T}_l(i), T_b(i))$, the set of trees in $\boldsymbol{T}_l(i)$ that overlap with $T_b(i)$ on R_j. $\Pi_i(\boldsymbol{T}_l(i), T_b(i))$ and $\Pi_j(\boldsymbol{T}_l(i), T_b(i))$ generally are not identical. Similar to step (4) in the wake-up procedure, R_j combines the routing entries for the trees in $\Pi_j(\boldsymbol{T}_l(i), T_b(i))$ with $T_b(i)$'s entry into a single logical entry.
2. We denote $\boldsymbol{T}_X(j)$ as the set of trees in $\boldsymbol{T}(j)$ that also belong to $X_i(\boldsymbol{T}_l(i), T_b(i))$. Therefore, if R_j belongs to some trees in $X_i(\boldsymbol{T}_l(i), T_b(i))$, i.e., $\boldsymbol{T}_X(j)$ is not empty, R_j is able to forward the traffic on $\boldsymbol{T}_X(j)$ also to $T_b(i)$ by "turning on" the respective interface to $T_b(i)$. In this case R_j performs the following.
   2.1 R_j modifies, if not done so, the sets of outgoing interfaces in each entry for the trees in $\boldsymbol{T}_X(j)$ so that $T_b(i)$ could also receive those traffic.
   2.2 Then R_j sends a $CONFIRM(\boldsymbol{T}_X(j), T_b(i))$ message back to R_i on $T_b(i)$ and it also sets $X_j(\boldsymbol{T}_l(i), T_b(i))$ to $X_i(\boldsymbol{T}_l(i), T_b(i)) - \boldsymbol{T}_X(j)$. That is, the trees in $\boldsymbol{T}_X(j)$ can be switched to $T_b(i)$.
   2.3 If $X_j(\boldsymbol{T}_l(i), T_b(i))$ and $\Pi_j(\boldsymbol{T}_l(i), T_b(i))$ are not both empty, R_j sends a $SWITCH(\Pi_j(\boldsymbol{T}_l(i), T_b(i)), X_j(\boldsymbol{T}_l(i), T_b(i)), T_b(i))$ message to upstream. A timer is also set for receiving confirmation for this request. If both $X_j(\boldsymbol{T}_l(i), T_b(i))$ and $\Pi_j(\boldsymbol{T}_l(i), T_b(i))$ are empty, this implies that the tree switching request is completely fulfilled and there is no need to send a *SWITCH()* message upstream.
3. If R_j does not belong to any trees in $X_i(\boldsymbol{T}_l(i), T_b(i))$, the router repeats step (2.3) with $X_j(\boldsymbol{T}_l(i), T_b(i)) = X_i(\boldsymbol{T}_l(i), T_b(i))$.

Based on the above, it is possible that each upstream router could partially fulfill the original tree switching request. Thus the set $X_i(\boldsymbol{T}_l(i), T_b(i))$ becomes smaller in size as the *SWITCH()* message travels upstream. It is also possible that none of the upstream routers could satisfy the tree switching requests and the RP will be responsible for discarding the message.

3.3. Confirmation procedure

The *CONFIRM()* message serves two functions. The first one is to build routing states for the trees to be switched to $T_b(i)$. Thus, when an interior router receives a *CONFIRM*($\boldsymbol{T}_X(j)$, $T_b(i)$) message on $T_b(i)$, it will create a new logical routing entry for the trees in $\boldsymbol{T}_X(j)$ and $T_b(i)$ if the trees in $\boldsymbol{T}_X(j)$ are overlapped with $T_b(i)$ or for the trees in $\boldsymbol{T}_X(j)$ otherwise.

The second function is to notify R_i that the tree switching request is either fully or partially satisfied. Upon receiving this message, R_i moves from the *Wait* state to the *Confirm* state and it performs the following.

1. Send a *PRUNE*(T_r) message for each $T_r \in \boldsymbol{T}_X(j)$ upstream on the tree.
2. Remove the routing entries for the trees in $\boldsymbol{T}_X(j)$.
3. Create a new logical routing entry for the trees in $\boldsymbol{T}_X(j)$ and $T_b(i)$ if the trees in $\boldsymbol{T}_X(j)$ are overlapped with $T_b(i)$ or for the trees in $\boldsymbol{T}_X(j)$ otherwise.

3.4. Pruning procedure

The *PRUNE()* message is for pruning the tree branch that has already been switched to the base multicast tree. Since pruning starts from the leaf routers, the *PRUNE()* message travels upstream on the tree. Any routers, upon receiving the message on a tree T, are required to delete the interface, where the message is received, from the T's entry. If the resulted set of outgoing interfaces is empty and the router is an on-tree router, the routing entry will be removed and the *PRUNE()* message will be forwarded further upstream.

3.5. Tree join and leave

The discussion on the TSP's operation so far assumes that the multicast trees under consideration do not change throughout the protocol operation. Nevertheless, multicast trees do change in size and shape in the real-world environment. With a new router added to a multicast tree, a designated router may no longer be a leaf router and problems may arise when the TSP performs pruning later on. To handle this situation, a designated router, which initiated a tree switching request and the process has not been completed yet, will suspend any tree join requests until all the outstanding tree switching processes are completed. On the other hand, the leaving of a designated router will affect on-going tree switching processes initiated by this router. Similar to the case of tree join, any tree leave requests from a designated router will be delayed until all the outstanding tree switching requests initiated by this router are completed.

4. Performance evaluation

In this section, we will evaluate the performance of the TSP when applied to PIM-SM. We will only consider the case of shared trees. The source-specific trees will not be considered as they can be treated similarly. Specifically, we will use simulation to evaluate the effect of TSP in the areas of tree overlapping, state reduction, and packet delay in terms of hop counts. Lastly, we will evaluate the impact of base tree selection method on the aforementioned performance areas.

In our simulation experiments, we generate a random graph with *100* nodes using the algorithms described in [6]. We set the average nodal degree to four and the cost of each link is assumed to be unity. We then select the designated nodes and RP randomly from the graph for each multicast tree. By selecting a shortest-path from a designated router to the RP, a shared tree is thus formed. Each data point in our simulation results is an average over *100* independent simulation experiments. When a designated router R_i starts the TSP, it selects the base multicast tree randomly from the set $\boldsymbol{T}(i) - \boldsymbol{T}_f(i)$ if the set is not empty; otherwise it selects randomly from $\boldsymbol{T}_f(i)$.

4.1. Degree of tree overlapping

We first define a metric for measuring the *degree of tree overlapping* for a set of shared trees $\boldsymbol{T}$ in (1).

$$DTO(\boldsymbol{T}) = \frac{\sum_{all\ R_i \in \boldsymbol{R}} \left(\frac{\eta_i - m_i}{n_i} \times \frac{N(\boldsymbol{T})}{N(\boldsymbol{T}) - 1} \right)}{N(\boldsymbol{R})}, \qquad (1)$$

where $DTO(\boldsymbol{T})$ is the degree of tree overlapping for $\boldsymbol{T}$, which ranges from 0 to 1. $\boldsymbol{R}$ is the set of routers that are on at least a tree in $\boldsymbol{T}$. $N(\boldsymbol{T})$ and $N(\boldsymbol{R})$ are the total number of multicast trees in $\boldsymbol{T}$ and the total number of routers in $\boldsymbol{R}$ respectively. For a router R_i,

1. n_i is the number of trees that this router belongs to.
2. η_i is the number of routing entries that are combined into logical entries.
3. m_i is the number of logical entries after combining the routing entries with the same sets of incoming and outgoing interfaces.

Thus, R_i does not contribute to the $DTO(\boldsymbol{T})$ if the trees are not overlapped on R_i. On the other hand, if R_i is on all the trees in $\boldsymbol{T}$ and they are all overlapped with each other, R_i contributes $1/N(\boldsymbol{R})$ to the $DTO(\boldsymbol{T})$. In a special case, where all the trees in $\boldsymbol{T}$ are completely overlapped, $DTO(\boldsymbol{T})$ is equal to *1*.

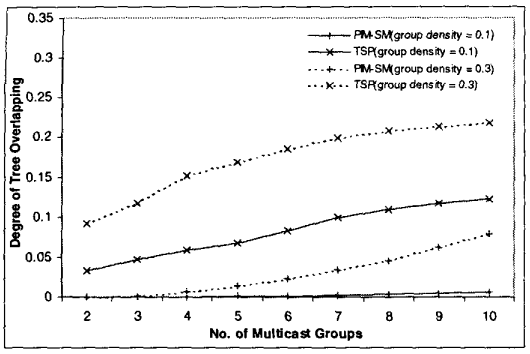

Figure 2. TSP and degree of tree overlapping.

We show the *DTO* under two cases in Fig. 2, where the group densities are given by *0.1* and *0.3*, and the number of multicast trees ranges from *2* to *10*. The group density of a multicast group is defined as the fraction of routers on the corresponding multicast tree that are designated routers. Without the TSP, the *DTO* increases with the number of multicast trees, as expected. With a higher group density, the rate of increase in *DTO* is higher since on average there is a higher probability for the trees to be overlapped on a router. When we apply the TSP to both cases, the *DTO* increases even further; however it is clear that the increase for the group density of *0.3* is approaching a plateau earlier than the other case. We expect the two curves for the group density of *0.3* will meet each other when the number of group increases to a point, where the original forest of trees has already had a significant overlapping among themselves and therefore little is gained by applying the TSP.

4.2. Multicast state reduction

We measure the state reduction according to (2).

$$SR(\boldsymbol{T}) = \frac{E_P(\boldsymbol{T}) - E_S(\boldsymbol{T})CR}{E(\boldsymbol{T})}, \qquad (2)$$

where $SR(\boldsymbol{T})$ is the percentage of state reduction for a set of trees $\boldsymbol{T}$. $E(\boldsymbol{T})$ is the total number of routing entries required for the multicast trees in $\boldsymbol{T}$. After executing the TSP on $\boldsymbol{T}$, $E_P(\boldsymbol{T})$ is the number of routing entries pruned away and $E_S(\boldsymbol{T})$ is the number of new routing entries added to the base multicast trees, as part of logical entries. Since the switched tree branches must be overlapped with at least the base tree, the states for those entries can be further reduced by performing state compression.

State compression is able to aggregate several routing entries sharing the same incoming and outgoing interfaces into a single "logical" entry. The actual storage requirement for such a logical entry is generally more than that for a single routing entry for a multicast group. One way of reducing the storage requirement for the logical entry is to "compress" redundant information like the incoming and outgoing interfaces. Nevertheless, there are still other state information about multicast groups, such as the soft state timer value, need to be retained for each individual group. Moreover, the actual storage reduction in the multicast state also depends on the internal implementation details. Thus it is not straightforward to compute the actual amount of state reduction using the state compression approach. Instead, we will present simulation results on the multicast state reduction under different state compression ratios for overlapped trees.

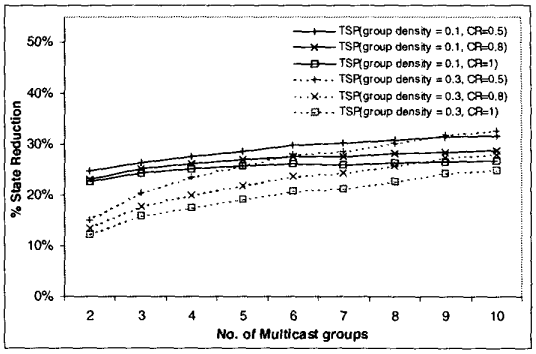

Figure 3. TSP and multicast state reduction.

Now we denote the compression ratio as *CR* and we consider *CR = 0.5, 0.8,* and *1* in Figure 3. The storage for each new entry added to the base trees is therefore discounted by *CR*. The case of *CR = 1* is equivalent to no state compression at all. Therefore, the amount of state reduction in this case is obtained by the difference between $E_P(\boldsymbol{T})$ and $E_S(\boldsymbol{T})$.

We summarize in the following several important points observed from the results in Figure 3.

1. The TSP could reduce *25%* of the multicast state for the group density of *0.1* even without state compression for overlapped trees. With state compression, the total state could be reduced by an additional 2-5%.
2. The TSP does not reduce as much state information for the group density of *0.3* without state compression for the overlapped trees. However, the rate of state reduction increases more steeply with the number of multicast groups. With the state compression, the states could be reduced by an additional *2-8%*, a wider margin than the case of *0.1*.
3. With the same $E_P(\boldsymbol{T})$ and $E_S(\boldsymbol{T})$, the state reduction for each case of group density approaches to the

same value when the number of multicast group increases.

In other words, the state reduction for a sparse shared tree (group density of *0.1*) is mainly gained from switching trees to the base multicast trees. Based on (2), this implies that the average "length" of a pruned tree branch is longer than the average "length" of a tree branch switched to a base tree. This is reasonable since the number of on-tree routers between a leaf router and the closest designated router upstream is higher for a sparse tree. As a result, more on-tree routers will be pruned away after tree switching. And the additional gain from compressing the states for overlapped trees is not significant.

When the trees are more bushy (group density of *0.3*), the contribution from tree switching is less but that from the state compression for overlapped trees increases. This is because the number of on-tree routers to be pruned away is less when there are more designated routers on the tree. On the other hand, there is already a large degree of tree overlapping when the group density is high; thus state compression make a more significant contribution to the state reduction in this case.

4.3. End-to-end delay

The TSP has an adverse effect on the end-to-end delay in terms of hop counts. For each multicast tree, we randomly pick *5* nodes in the network as the designated routers for source hosts (first-hop routers) and we measure the average hop count from the first-hop routers to other designated routers on the tree as the destinations. The first-hop routers will send packets to the RP first and then the RP will forward them to other designated routers. From Figure 4, we observe an average increase of *1* hop in the end-to-end delay and the average end-to-end delay without tree switching is *6* hops. And the increase is not sensitive to the number of multicast groups and the group density. The increase in hop count is due to a longer path from the RP to the designated routers since the tree switching will likely move the designated routers further away from the RP.

4.4. Protocol overhead

The protocol overhead depends on both the degree of overlapping and the timer values. More designated router are expected to initiate the protocol when the degree of tree overlapping is high. While we cannot control the network environment, we could tune the protocol through the various timers. The timer that triggers the transition from the *Idle* state to *Wake-up* state clearly has a direct impact on the frequency of executing the protocol. One approach is to employ exponential backoff for updating the timer value when the previous tree switching request is unsuccessful. In this way, the protocol overhead will be reduced but the protocol may react slowly to the network dynamics.

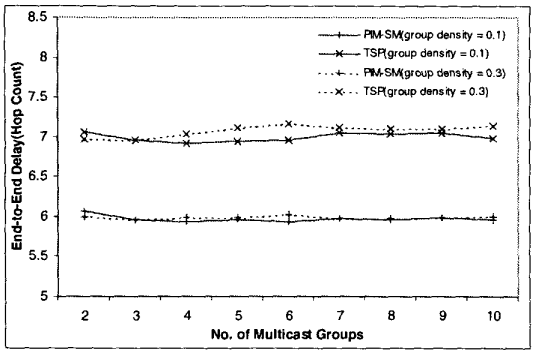

Figure 4. TSP and end-to-end delay (hop count).

5. Conclusions

In this paper we have proposed a new Tree Switching Protocol (TSP) for reducing the total state requirement for a forest of shared trees or a forest of source-specific trees. This protocol exploits overlapping among multicast trees by switching multicast trees to a base multicast tree. Simulation results show that a significant state reduction could be obtained for a sparse tree through tree switching. The TSP is a loop-free and very efficient distributed protocol.

References

[1] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C. Liu, and L. Wei, "The PIM Architecture for Wide-area Multicast Routing," *IEEE/ACM Trans. on Networking*, vol. 4, no. 2, 1996.

[2] M. Handley, "Session Directories and Scalable Internet Multicast Address Allocation," in *Proc. of SIGCOMM*, pp. 105-117, 1998.

[3] P. Tsuchiya, "Efficient and Flexible Hierarchical Address Assignment," TM-ARH-018495, Bellcore, February 1991.

[4] R. Briscoe and M. Tatham, "End to End Aggregation of Multicast Addresses," *Internet Draft (draft-briscoe-ama-00.txt)*, Nov. 1997.

[5] P. I. Radoslavov, D. Estrin, and R. Govindan, "Exploiting the Bandwidth-Memory Tradeoff in Multicast State Aggregation", *Tech. Rep. 99-697, USC Computer Science Department*, 1999.

[6] B.M. Waxman, "Routing of Multipoint Connections," *IEEE J. Sel. Areas in Communications*, vol. 6, no. 9, pp. 1617-1622, 1988.