

A Simple and Fast Distributed Algorithm to Compute a Minimum Spanning Tree in the Internet*

H. Abdel-Wahab, I. Stoica, F. Sultan and K. Wilson

Department of Computer Science

Old Dominion University

Norfolk, Virginia, 23529

e-mail: {wahab, stoica, sultan, ksw}@cs.odu.edu

Summary

A central problem in wide area networks is to efficiently multicast a message to all members (*hosts*) of a target group. One of the most effective methods to multicast a message is to send the message along the edges of a spanning tree connecting all the members of the group. In this paper we propose a new fully distributed algorithm to build a minimum spanning tree (MST) in a generic communication network. During the execution, the algorithm maintains a collection of disjoint trees spanning all the group members. Every tree, which initially consists of only one node, *independently* expands by joining the closest tree, until all the nodes are connected in a single tree. The resulting communication topology is both robust (there are no singularities subject to failures) and scalable (every node stores a limited amount of local information that is independent of the size of the network).

Key Words: Minimum-Spanning Trees, Multicast, Communication Networks, Internet, Distributed Algorithms

*This work is supported by the National Science Foundation Grant # NCR-9313857

1 Introduction

The problem of building spanning trees in a communication network is commonly addressed in relation with designing efficient multicasting algorithms in wide area networks. The overall bandwidth required by a multicast operation is the main measure of its efficiency; using a tree for routing multicast messages to members of a target group aims at reducing the number of messages needed for a message originating at a given node to be received at all the other recipient nodes.

In general, wide-area network applications (e.g., computer supported conferencing [1], Internet audio and video multicast [13] etc.) employing sharing of information by agents located throughout an internetwork and belonging to disjoint logical groups have to consider the problem of efficiently propagating information to members of the same group. Different patterns of traffic and traffic requirements are displayed by various applications. This involves optimizing the bandwidth consumption as the first goal (by reducing the number of messages generated by a multicast delivery scheme), but at the same time must take into account the problem of minimizing the transmission delays and/or the incurred transmission costs over the communication links. An optimal multicast delivery tree, that is a tree of minimum sum over the weight of its edges in a metric based on delay should be used for routing messages within one given group. Capacity constraints can be further added to account for the limited capacity of the links relative to the traffic bandwidth demands [16].

This paper describes a distributed algorithm for computing the minimum spanning tree (MST) in a generic communication network modeled as a connected graph, considering as metric the transmission delay between adjacent nodes. Building the MST is based on the assumption that a node can determine the round trip time of a message sent to an adjacent node. The algorithm exploits Bollobos idea [5] of maintaining at all times during the execution a collection of disjoint trees (forest) spanning all the nodes in the initial graph; trees in the forest expand by being merged through edges of minimum weight until all the nodes are connected in a single tree, which is the final MST.

Using other metrics for the edge weights is straightforward, while traffic characteristics and capacity constraints are not taken into account: we assume for the intended traffic that the bandwidth requirements are less than the link bandwidth. However, since the traffic requirements can be deemed as characteristic to a specific group, the MST would eventually be reconfigured during the execution of a group management algorithm so as to take into account the limited capacity of the links in the presence of multiple multicasts with high bandwidth requirements.

The remainder of the paper is organized as follows. Section 2 describes the MST algorithm in generic terms, while section 3 gives the details of the algorithm: the initialization phase, the construction phase, an example and its message complexity analysis. Section 4 briefly discusses and presents the measurements of our algorithm implementation. Section 5 is a recent survey for related work. Finally, in section 6 we state our conclusions and we list some extensions to our work.

- | |
|---|
| <ol style="list-style-type: none"> 1. for every vertex $v_i \in G$ 2. $T_i \leftarrow \{v_i\}$ 3. until all vertices are connected 4. for every T_i do in parallel 5. pick the edge e_l of the smallest weight that connects T_i with another sub-graph T_k 6. $T_{\min(i,k)} \leftarrow T_i \cup T_k \cup \{e_l\}$ |
|---|

Figure 1: The generic MST algorithm

2 The Generic MST Algorithm

Let $G = (V, E)$ denote a connected, undirected graph, where V represents the set of vertices, and E represents the set of edges. Further, assume that every edge in E has associated a weight, and no two weights are equal. The generic algorithm to construct a MST is shown in Figure 1. The algorithm starts by creating $|V|$ distinct sub-graphs¹, each sub-graph containing one vertex (steps 1, 2). Next, every sub-graph, independently, chooses the edge of the minimum weight that connects it with another sub-graph, and the two sub-graphs are joined along that edge (steps 4-6). Steps 4-6 are repeated until all vertices are connected.

To prove the correctness of this algorithm we need to show that during the construction no cycles appear and the resulting graph is indeed the MST.

Lemma 1 *The algorithm described in figure 1 does not generate cycles.*

Proof. Initially (steps 1, 2), every sub-graph contains only one vertex and consequently there are no cycles.

For steps 3-4 assume that no sub-graph T_i contains cycles. Next suppose that a subset of sub-graphs $T_{i_1}, T_{i_2}, \dots, T_{i_k}$ choose at the same time connecting edges (i.e., T_{i_1} chooses e_{i_1} , T_{i_2} chooses e_{i_2} , \dots , T_{i_k} chooses e_{i_k}) such that a cycle involving vertices in $T_{i_1}, T_{i_2}, \dots, T_{i_k}$ is created. Since every sub-graph T_{i_j} ($1 \leq j \leq k$) chooses only one connecting edge and there are at least k edges required to create a cycle involving vertices in $T_{i_1}, T_{i_2}, \dots, T_{i_k}$, it is clear that all e_{i_j} must be different, i.e., there are not two sub-graphs that choose the same edge. For simplicity let us assume that e_{i_1} connects T_{i_1} to T_{i_2} , e_{i_2} connects T_{i_2} to T_{i_3} , \dots , e_{i_k} connects T_{i_k} to T_{i_1} (otherwise we can easily relabel the sub-graphs and edges). Since there are no two edges with the same weight this implies that $e_{i_1} < e_{i_k}$ (otherwise, at the step 5, T_{i_1} would have chosen e_{i_k} instead of e_{i_1}), $e_{i_2} < e_{i_1}$, \dots , $e_{i_k} < e_{i_{k-1}}$. But from the last $k - 1$ inequalities we obtain $e_{i_k} < e_{i_1}$, which contradicts the first inequality and therefore our assumption that a cycle can be created is false.

Thus the final sub-graph that connects all vertices of G is a spanning tree. \square

Further, due to Bollobas [5], we have the following result:

Theorem 1 *The spanning tree constructed by the algorithm described in Figure 1 is a MST of G . Moreover the resulting MST is unique.*

¹In fact, as we will show in Lemma 1, the sub-graphs generated by algorithm 1 are trees.

3 The MST Algorithm in a Communication Network

We model the communication network as a connected undirected graph $G = (V, E)$, where the set of nodes V represents the set of workstations (or processors), and the set of edges E represents a sub-set of *virtual* interconnections. We say that two workstations (nodes) are virtually interconnected, if there is a communication path along which they can exchange messages. To each edge we associate a weight that consists of the round trip time (RTT) between the two end nodes.

The MST algorithm is divided in two phases: *initialization* and *construction*. The *initialization* phase computes the RTTs for each edge in the graph and makes sure that all the weights are unique.² The *construction* phase implements the generic algorithm presented in Figure 1. The following two sections present the details of the two phases.

3.1 Initialization Phase

For simplicity, we assume that the MST is built as a result of an external request and no more than one request is received at a time by any node in the graph³, i.e., the MST construction starts from a single node, called **MST initiator**. During this phase every node manages the following data structures:

- *adj_list* - contains the identifiers of all neighbors;
- *rtt_list* - contains the pairs consisting of the RTT to each neighbor, and the identifier of the node at which the RTT was computed.

The initialization phase of the MST algorithm is shown in Figure 2. First, the initiator node sends a START_MST message to itself. Further all the nodes, including the initiator, execute the same code. Upon receiving the START_MST message for the *first time*, a node forwards it to all its neighbors (all nodes from its *adj_list*) excepting the sender node, and starts to compute the RTT to every neighbor with a smaller identifier. In this way the weight of every edge is computed only once, namely by its end node with the largest identifier.⁴ Once the RTT of an edge is computed, it is inserted, along with the node that has computed it, in the *rtt_list* of both end nodes by using a RTT_MST message. After the weights of all incident edges are computed the node starts the second phase of the algorithm.

To ensure a total order relation between the edge weights we define the following relation between any two elements in *rtt_list*:

$$(RTT_i, n_i) < (RTT_j, n_j) \equiv ((RTT_i < RTT_j) \text{ or } ((RTT_i = RTT_j) \text{ and } (n_i < n_j))) \quad (1)$$

²This is achieved by associating to each weight the identifier of the node that computes the corresponding RTT (see Section 3.1).

³Our current implementation supports multiple requests as well. However, the implementation details are not relevant for the algorithm, and therefore are omitted.

⁴This is because, in a real network, allowing both end-nodes to independently compute the RTT of their edge may result in different values.

```

protocol message types
  START_MST = [...], RTT_MST = [rtt, ...]
variables
  adj_list, rtt_list, first_received
initialization
  first_received := FALSE
send START_MST to local_id // send the message to itself
loop-forever
  on receiving message
  case START_MST:
    if (first_received = FALSE)
      first_received := TRUE // received START_MST first time
      for every n in adj_list except sender_id
        send START_MST to n
      for every n in adj_list
        if n < local_id
          compute the RTT to n
          insert (RTT_MST.rtt, local_id) in rtt_list
          RTT_MST.rtt := RTT
          send RTT_MST to n
  case RTT_MST:
    insert (RTT_MST.rtt, sender) in rtt_list
    if the weights of all incident edges have been computed
      execute MST construction phase (Figure 4)

```

Figure 2: The initialization phase of the MST algorithm

3.2 MST Construction Phase

This phase implements the MST generic algorithm (see Figure 1). Initially, the algorithm creates a set of disjoint trees, each tree containing one node (Figure 1, steps 1-2). Each of these trees is labeled with the identifier of its node (here, we assume that every node has a unique identifier). Next, every tree searches independently the edge of the minimum weight, called *minimum bridge*, that connects it with another tree (Figure 1, steps 4-5). If such an edge exists, the two trees are joined and the new tree is labeled with the smallest label of the initial trees (Figure 1, step 6). After joining, the root of the tree with the smallest label becomes the root of the new tree. Since initially every single-node tree is labeled with the identifier of its node, it is easy to see that the label of any new tree is the smallest identifier among all nodes in the tree, and the root of the tree is the node with the smallest identifier (therefore, the label of the tree is the identifier of its root). This procedure is repeated until all nodes are connected.

To compute the *minimum bridge* of a tree, we use a message called COMPUTE. This message contains two fields:

- *label* - the label of the tree;

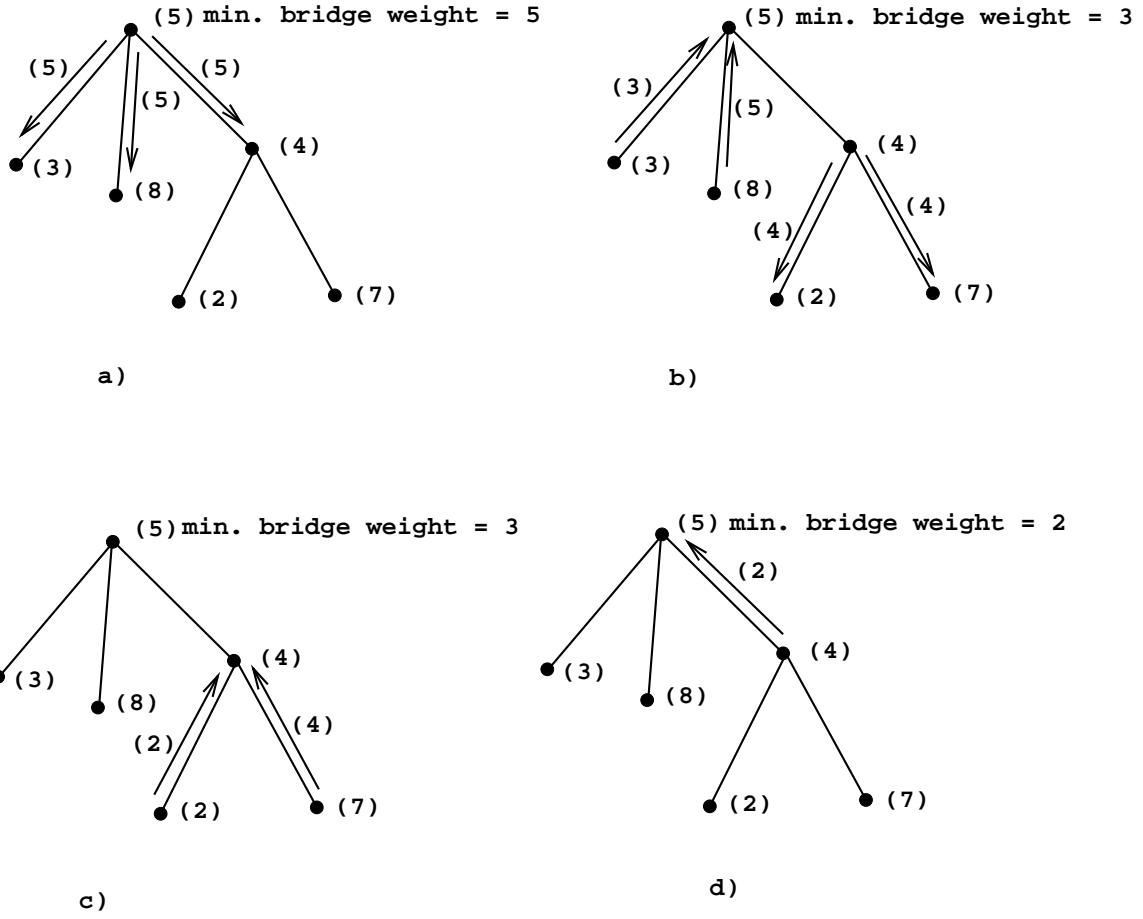


Figure 3: The computation of the minimum bridge weight. Associated to every node (in parenthesis) is the smallest weight of an incident edge that connects it with another tree (*loc_min_bridge_wt*). Associated to every message is the value contained in *min_bridge_wt* field.

- *min_bridge_wt* - used to compute the *minimum bridge* weight.

In addition, every node manages the following data structures:

- *label* - the label of the tree to which the node belongs;
- *cand_mst_heap* - contains the identifiers of all neighbors (in graph G) which belong to other trees. These neighbors are stored in the increasing order of their corresponding weights. More precisely, given the set of neighbors n_1, n_2, \dots, n_m , of node v_i , which do not belong to the same tree as v_i , then the *cand_mst_heap* of v_i is the sorted sequence of neighbors: $n_{i_1}, n_{i_2}, \dots, n_{i_m}$, such that $weight(v_i, n_{i_1}) < weight(v_i, n_{i_2}) < \dots < weight(v_i, n_{i_m})$ (where ' $<$ ' is defined by (1));
- *loc_min_bridge_wt* - the smallest weight of an incident edge that joins the current node with a node in another tree (i.e., the first node in *cand_mst_heap*). If there is no such an edge, then *loc_min_bridge_wt* is set to ∞ . Notice that the *minimum bridge* weight is the minimum among all *loc_min_bridge_wt* in the tree;

```

protocol message types
COMPUTE = [min_bridge_wt, label, ...]
variables
    adj_list, mst_adj_list, loc_label, loc_min_bridge_wt, cand_mst_heap, min_chld_bridge_wt
initializations
    mst_adj_list :=  $\emptyset$  // contains the node neighbors in MST
    cand_mst_heap := sort_by_wt(adj_list)
    loc_label := loc_id
wait for all neighbors to finish initialization phase
LABEL 1:
if node is root
    if (mst_adj_list =  $\emptyset$ ) // the tree consists of only one node
        add_MST_edge(mst_adj_list, cand_mst_heap)
    COMPUTE.label := loc_id
    COMPUTE.min_bridge_wt := get_min_wt(cand_mst_heap)
    send COMPUTE to all children
on receiving message
    case COMPUTE: // Computation sub-phase
        See Figure 4 for COMPUTE message handling.
    case DIFFUSE: // Difussion sub-phase
        if (DIFFUSE.label := loc_label)
            if (DIFFUSE.min_bridge_wt := loc_min_bridge_wt)
                add_MST_edge(mst_adj_list, cand_mst_heap) // min. bridge found
            else
                send DIFFUSE to all children

```

Figure 4: The MST algorithm.

- *mst_adj_list* - contains the identifiers of all neighbors that belong to the same tree. Thus, the union of all nodes in *cand_mst_heap* and all nodes in *mst_adj_list* represent the set of all neighbors of the current node in graph G .

The computation of the *minimum bridge* weight is based on the diffusing computation paradigm as proposed by Dijkstra and Scholten [10]. The root sends a COMPUTE message with the *min_bridge_wt* initialized to its *loc_min_bridge_wt* (i.e., the smallest weight of an edge that joins the root with a node from another tree). On receiving this message, each node checks whether COMPUTE.*min_bridge_wt* is smaller than its *loc_min_bridge_wt*. If this is the case, then it updates COMPUTE.*min_bridge_wt* to *loc_min_bridge_wt*, and forwards the message to its children, if any. If the node is a leaf, then it sends the message back to its parent. Upon receiving the replies back from all children, every node computes the minimum among all *min_bridge_wt* fields in the received messages and sends it to its parent. Finally, when the root gets all the replies, it computes the *minimum bridge* weight of the tree. To clarify the ideas, a simple example is shown in Figure 3.

The complete operations, performed by every node during the computation of the *minimum*

```

if (COMPUTE.label  $\leq$  loc_label)
  if (COMPUTE.label < loc_label)
    loc_label := COMPUTE.label
    establish the sender as the new parent
  if received from parent
    loc_min_bridge_wt := get_min_wt(cand_mst_heap)
    if (COMPUTE.min_bridge_wt > loc_min_bridge_wt)
      COMPUTE.min_bridge_wt := loc_min_bridge_wt
    if node is leaf
      send COMPUTE to parent
    else
      send COMPUTE to all children
      min_chld_bridge_wt :=  $\infty$ 
  else // received from child
    min_chld_bridge_wt := min(min_chld_bridge_wt, COMPUTE.min_bridge_wt)
  if received replies from all children
    if node is root
      if (COMPUTE.min_bridge_wt = loc_min_bridge_wt)
        if (COMPUTE.min_bridge_wt =  $\infty$ )
          exit // MST construction is completed
          add_MST_edge(mst_adj_list, cand_mst_heap)
        else
          DIFFUSE.label = COMPUTE.label
          DIFFUSE.min_bridge_wt = COMPUTE.min_bridge_wt
          send DIFFUSE to all children
        go to LABEL 1 (Figure 4)
      else // the node is not a root
        COMPUTE.min_bridge_wt := min_chld_bridge_wt
        send COMPUTE to parent

```

Figure 5: The COMPUTE message handling (computation sub-phase).

bridge, are given below (the detailed algorithm is depicted in Figures 4 and 5):

1. The *root* initializes the COMPUTE.*min_bridge_wt* field to its *loc_min_bridge_wt* and the COMPUTE.*label* field to its identifier. Next, the *root* sends the message to its children and waits for replies.
2. Upon receiving from its parent a COMPUTE message with the *label* field equal to its label, the current node checks whether the COMPUTE.*min_bridge_wt* is greater than its *loc_min_bridge_wt*. If this is the case, then the COMPUTE.*min_bridge_wt* is updated to *loc_min_bridge_wt*. Further, the node forwards the message to its children and waits for replies. If the node does not have any children (i.e., it is a *leaf*), then it sends the COMPUTE message back to the parent;
3. Upon receiving from any of its neighbors a COMPUTE message with the *label* field less than its label, the current node assumes that the sender is its *new parent*. Consequently, it updates its label to COMPUTE.*label* and continues with operation 2;
4. Every COMPUTE message with the *label* field greater than the node's label is ignored.
5. Upon receiving all COMPUTE messages from its children, each node computes the minimum *min_bridge_wt* among all the received replies and sends the result to its parent. If the current node is the root of the tree then it checks if the received *min_bridge_wt* is equal to *loc_min_bridge_wt*. If this is the case, (i.e., *minimum bridge* is among its incident edges) the root joins the current tree with the other tree along *minimum bridge*. Otherwise, it sends out a DIFFUSE message containing the computed *min_bridge_wt* to all its children.

Notice that the above operations also perform the node relabeling task. As an example, consider two trees T_i and T_j that have just been connected by the edge (v_{i_k}, v_{j_l}) , where $v_{i_k} \in T_i$ and $v_{j_l} \in T_j$. Further, suppose $label(T_i) \leq label(T_j)$. If v_{j_l} forwards the T_j 's COMPUTE message to v_{i_k} , then v_{i_k} ignores the message since in this case COMPUTE.*label* ($= label(T_j)$) is greater than its label ($= label(T_i)$). On the other hand, if v_{i_k} forwards the T_i 's COMPUTE message, then v_{j_l} updates its label to the *label* field received in the message and forwards the message to its children (operation 3) that, in turn, execute recursively the same operation. This ensures that every node from T_j will update its label to T_i 's *label*, thus becoming a node of T_i . Also, notice that when T_j 's root receives for the first time the T_i 's COMPUTE message it becomes a child of the sender and will no longer generate any other COMPUTE message for T_j . Thus, T_j becomes a subtree of T_i , which completes the join operation.

Once the root has computed the weight of the *minimum bridge*, and this is not equal to *loc_min_bridge_wt* (operation 5), it initiates the sending of a DIFFUSE message to all its children (Figure 4). This message consists of the same fields as COMPUTE message : *label* and *min_widge_wt*. The *label* field contains the tree label, while the *min_bridge_wt* contains the *minimum bridge* weight of the tree previously computed. On receiving this message, every node checks

whether the `DIFFUSE.min_bridge_wt` is equal to its `loc_min_bridge_wt`. If this is the case, then this means that the *minimum bridge* is among the current node's incident edges. As a result the current tree is joined with the other tree along the *minimum bridge*.

After the root sends the `DIFFUSE` message, it starts the computation of the *minimum bridge* weight for the new tree. Notice that, as we have shown before, after two trees were joined only the root with the minimum label finishes computing the new *minimum bridge* weight; the other root ends up by receiving a `COMPUTE` message with a smaller *label* and, consequently, becomes a simple node of the new tree.

The last issue that has to be addressed is how to detect the termination of the MST construction phase. Clearly, once the MST is constructed, at every node the `cand_mst_heap` becomes empty (since all the nodes belong to the same spanning tree). As a result, after MST is constructed, the weight of the *minimum bridge* becomes ∞ because no such a bridge exists. Therefore, the root of the MST naturally detects termination when the computed weight of the *minimum bridge* is ∞ .

When a new edge is added to the MST, the `add_MST_edge(mst_adj_list, cand_mst_search)` function is called by one of the two end-nodes of a *minimum bridge*. The purpose of this function is to join the two sub-trees along the *minimum bridge*. This is done in two steps: first, the caller deletes the neighbor identifier from the `cand_mst_heap` and inserts it in the `mst_adj_list`; second, a special message is sent to the neighbor along the *minimum bridge* edge. Upon receiving this message the neighbor inserts the sender identifier in its `mst_adj_list` and, at the same time, deletes it from the `cand_mst_heap`. In this way, each end-node of the *minimum bridge* inserts in its `mst_adj_list` the identifier of the other end-node. Notice that, since both two end-nodes may try to connect to each other at the same time, special caution must be taken to avoid insertion of duplicates in `mst_adj_lists`.

3.3 An Example

We use the graph from Figure 6 to illustrate the construction phase of our algorithm.

At the beginning of the construction phase of the algorithm, every node assumes that it is a root of a tree (consisting of that node) and it is labeled with its identifier. Next, every node searches for the incident edge with the smallest weight. In our example node 1 finds edge (1, 3), 2 finds (2, 3), 3 finds (3, 4) etc. Further, since the construction phase is asynchronous we have to pick, for the purpose of our example, an order in which trees are joined. This order is arbitrary, i.e., the reader can choose any other order without affecting the result.

Now, assume that node 4 is the first one that performs the joining (i.e., edge (3, 4) is added to the MST). As a result, after joining, node 4 initiates a computation sub-phase by issuing a `COMPUTE` message with `label = 4` (the root identifier) and `min_bridge_weight = 3` (i.e., the minimum weight of an incident edge that connects node 4 with a node in another tree; in this case node 5) and sending it to node 3. Since the label of 3 is less than the label it receives in the `COMPUTE` message, the message is ignored. In turn, node 3 chooses *independently* the

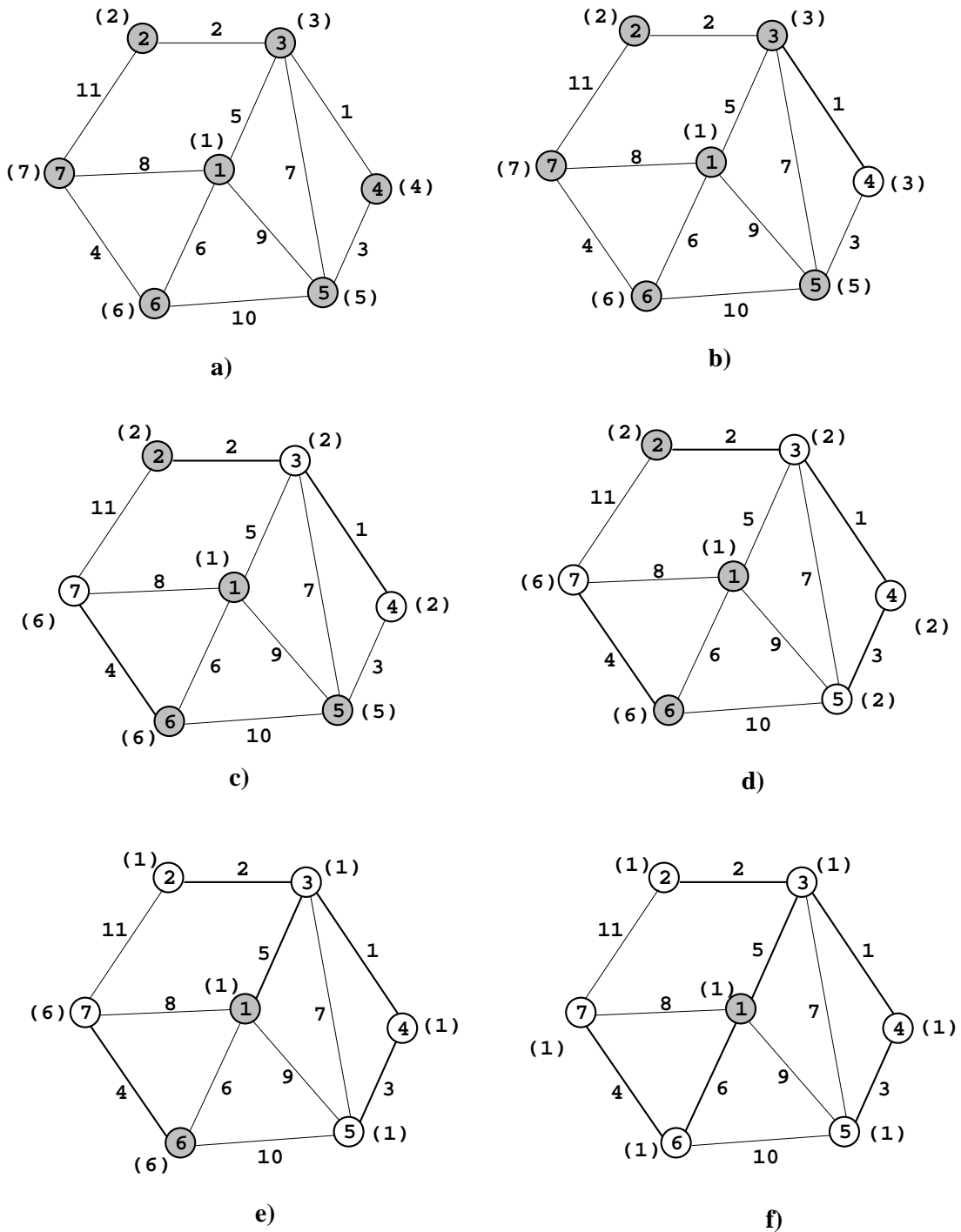


Figure 6: An example of a MST construction. The identifiers of the nodes and the weights of the edges are unique. The roots of the trees in different stages of the MST construction are represented by shaded circles. The labels of every node are written in parenthesis.

same edge (3,4) to join it with node 4. After joining, node 3 also sends a COMPUTE message to node 4, but with $label = 3$ and $min_bridge_weight = 2$. Upon receiving this message, node 4 updates its label to the value of the label received in the COMPUTE message and becomes the leaf in the new tree consisting of nodes 3 and 4 (Figure 6.b). Also, it is easy to see that since the $loc_min_bridge_weight$ of node 4 is 3, the min_bridge_weight field from the message is not modified and remains 2. Next, being a leaf of the new formed tree, node 4 sends back the COMPUTE to its parent. Upon receiving this message, node 3 (which is also the root of the tree) finds that the min_bridge_weight is 2 and thus is equal to its loc_min_bridge . Consequently, it decides that the edge (2,3) is the edge of a minimum weight (*minimum bridge*) that connects the tree consisting of nodes 3 and 4 with another tree and adds it to the MST. Notice that in this case there is no need to send any DIFFUSE message in order to find the *minimum bridge*.

Next, consider that node 2 chooses the edge (2,3), adds it to the MST and initiates a computation sub-phase by sending a COMPUTE message with $label = 2$ and $min_bridge_weight = 11$ to its closest neighbor, node 3. Since node 3 has label (3) which is greater than the label received in the COMPUTE message, it updates its label to 2, becomes a node of the new tree (having as root node 2) and forwards the message to all its neighbors in the MST, excepting the sender (i.e., node 4). At the same time, it updates the min_bridge_weight field from the message to its $loc_min_bridge_weight$ (5). Upon receiving the COMPUTE message, node 4 updates its label to 2 and the message min_bridge_weight field to 3. Next, node 4 (now a leaf of the new tree) sends back the COMPUTE message that further is forwarded by node 3 to the root, i.e., node 2. Upon receiving this message, the root finds that the min_bridge_weight field from the received message is not equal to its $loc_min_bridge_weight$ and therefore sends a DIFFUSE message back to its children. When node 3 receives this new message it checks whether or not the received min_bridge_weight field is equal to its $loc_min_bridge_weight$. Since this is not true, the message is forwarded to node 4 that finally finds that its $loc_min_bridge_weight$ is equal to the received min_bridge_weight field. Thus, it chooses the edge (4,5) to join its tree to the tree consisting of node 5.

In our example we assume that nodes 6 and 7 are joined before node 5 is joined to 4 (Figure 6.c.). After node 5 is joined to 4, during the procedure described above, the graph and the partially constructed MST are depicted in Figure 6.d. Finally, the last two edges (1,3) and (1,6) (Figures 6.e and 6.f) are added to the MST and the algorithm terminates.

3.4 Time Complexity Analysis

In this section we analyze the time-complexity of our algorithm. In our model, we assume that the propagation delay is at most one time unit and all messages, regardless the type contain $O(\log n)$ bits of information, where n is the number of nodes in the graph.

First, we derive the time-complexity result for the initialization phase.

Lemma 2 *Given a graph G with n nodes, the initialization phase takes $O(n)$ time.*

Proof. The proof is immediate. First, notice that after at most n time units the START_MST message is received by all the nodes in G (since the longest possible propagation path of a message in G is n). Next, every node has to compute the round trip times to all its neighbors. Since every node has at most $n - 1$ neighbors, and to compute the round trip delay to a node requires 2 messages, this task can be done in at most $2(n - 1)$ time units⁵. Therefore, the initialization phase takes at most $3n - 2$ time units. \square

The following three lemma determine the time complexity of the construction phase.

Lemma 3 *Given a sub-tree T of a graph G with n nodes, at most $O(n)$ COMPUTE/DIFFUSE messages are exchanged between nodes of T , until T is joined with another tree.*

Proof. Suppose that no COMPUTE message with a label different than T 's label is received by any node in the tree. During the *minimum bridge* weight computation every edge of the tree is traversed two times by the COMPUTE message: first, when the parent forwards the message to its children and second when the child sends its reply. Since for finding the *minimum bridge* at most $n - 1$ DIFFUSE messages (one per edge) are required, it follows that the *minimum bridge* can be determined using at most $3(n - 1)$ messages. Once the *minimum bridge* is determined, T is joined with the other tree along its *minimum bridge*.

Next, suppose that $v_i \in T$ receives a COMPUTE message with a label less than the T 's label. Consequently, v_i becomes the children of the sender and forwards the new message. Therefore v_i will no longer send a reply to its old parent in T . Thus, any computation sub-phase initiated by the T 's root at the same time or after at least one node in T receives a COMPUTE message with a label less than the T 's label cannot complete and consequently no other messages will be issued by T 's root. Therefore, in this case, we have again that after at most $3(n - 1)$ COMPUTE/DIFFUSE messages, T is joined with another tree.

Finally, if a COMPUTE message with a label greater than T 's label is received, then it is simply ignored. Thus, in all cases after at most $3(n - 1)$ COMPUTE/DIFFUSE messages, T is joined with another tree in G , which proves our lemma. \square

Lemma 4 *Given a sub-tree T of a graph G with n nodes, after $O(n)$ time-units, T is joined with another tree.*

Proof. Since the propagation time is assumed to be at most one time-unit, the COMPUTE/DIFFUSE messages exchanged, until T is joined with another tree, counts for at most $3(n - 1)$ time units.

Besides COMPUTE/DIFFUSE messages, there are two other situations in which messages are exchanged during the computation sub-phase.

First, upon receiving a COMPUTE message, every node computes the minimum weight of an incident edge that connects that node with other nodes (*loc_min_bridge_weight*). Remember

⁵Here, we assume that the round-trip time is sequentially computed. In practice it can be computed concurrently, as follows: first, the node sends the RTT_MST message to *all* its neighbors, and next it waits for receiving the acknowledgments.

that the neighbors that did not belong to the same tree last time when the *loc_min_bridge_weight* was computed are in the *cand_mst_list*. Since in the meantime it is possible that the closest neighbor in the *cand_mst_list* to be now in the same tree, the node verifies this by sending a message to its neighbor asking it for its label. If the neighbor and the node labels are different, then the *loc_min_bridge_weight* remains the same and the computation proceeds. In this case, for every COMPUTE message, at most two other messages are exchanged, one for asking the neighbor about its label and one for the reply. If *cand_mst_list* is empty (i.e., all neighbors are in the same tree), then no messages are exchanged and *loc_min_bridge_weight* = ∞ . On the other hand, if the label of the node and the label of the closest neighbor from *cand_mst_list* are equal this means that both nodes are in the same tree and therefore the neighbor is deleted from *cand_mst_list*. This procedure is repeated until a neighbor with a different label is found or the *cand_mst_list* becomes empty. Now, notice that the maximum numbers of neighbors that can be deleted from a *cand_mst_list* is exactly $n - 1$, i.e., the number of nodes in T . Since these deletions can be concurrently performed at every node of T , this counts for at most $2(n - 1)$ time units.

The second situation that implies message exchanging during the MST computation sub-phase is when the selected edge is actually inserted in the MST. More specifically, when a node selects an edge to be added to the MST it sends a message to the corresponding neighbor. As a result, every node inserts its corresponding neighbor in *mst_adj_list* (e.g., when edge (n_i, n_j) is added, node n_i inserts node n_j in its *mst_adj_list*, while n_j inserts n_i in its *mst_adj_list*). This task adds another 2 time-units and therefore the time interval between the moment when T is constructed and the moment when it is joined with another tree is at most $3(n - 1) + 2(n - 1) + 2 = 5n - 3$ time units, which proves the lemma. \square

Given a graph G , we define the *construction-tree* of G ($CT(G)$, for short) as follows:

- The nodes of $CT(G)$ represent the sub-trees generated during the MST construction.
- If T_1 and T_2 are two sub-trees that are joined during the MST algorithm in a sub-tree T , then T is the parent of T_1 and T_2 in $CT(G)$.

For simplicity, we label the CT 's nodes with the label of the sub-tree it represents. Figure 7 shows the CT for the example in Figure 6. Notice that the root of $CT(G)$ is the MST of G .

Lemma 5 *Given a graph G with n nodes, the construction phase takes $O(n)$ time.*

Proof. Let T be the root of $CT(G)$ and let T' and T'' be its children. Next, let $n_1 \leq n_2$, where n_1 and n_2 represent the number of nodes in T' and T'' respectively. From Lemma 1, after at most $5n_1 - 3$ time units (from the moment it was created), T' is joined with T'' . Let us associate to the edge (T', T) a weight equal to $5n_1 - 3$. Since $n = n_1 + n_2$, it is easy to see that the weight of (T', T) is $\leq 5(n/2) - 3$. Notice that the weight associated to (T', T) represents an upper bound for the time interval between the moment at which T' is created, and the moment at which T is created.

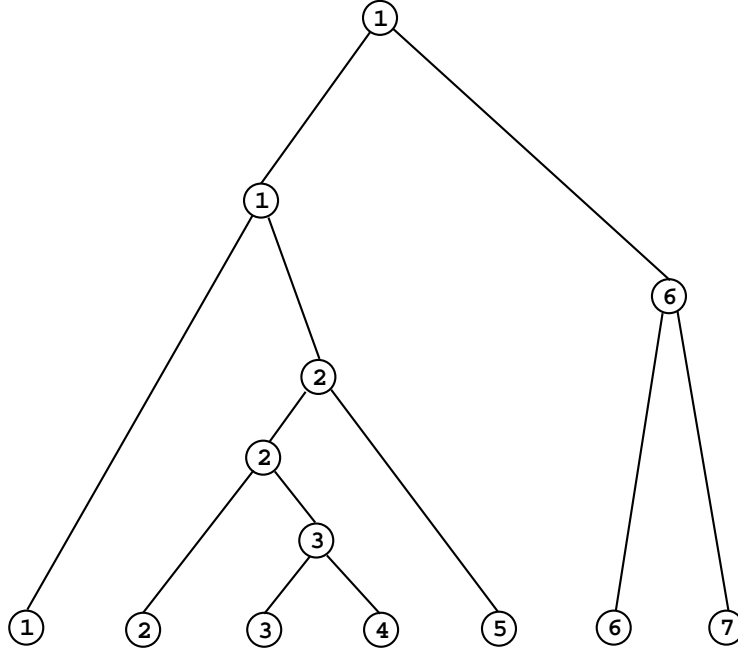


Figure 7: The construction tree (CT) for the example in Figure 6. The nodes are labeled with the label of the sub-tree it represents.

Further, using the same technique, we move recursively down the tree; from the current node T' we chose the child that represent the sub-tree T'_1 with the minimum number of nodes, and we associate to the edge (T', T'_1) the weight $5(n'/2) - 3$, where n' is the number of nodes in the sub-tree represented by T' . Next, the current node becomes T'_1 and the procedure continues. As a result, we obtain a path from T to a leaf T_l whose length represents an upper bound for the time interval between the moment at which T_l enters in the construction phase, and the moment when the last edge is added to the MST (i.e., the T is created). Thus, the upper bound for the entire time interval is given by:

$$\left(5\frac{n}{2} - 3\right) + \left(5\frac{n}{2^2} - 3\right) + \left(5\frac{n}{2^4} - 3\right) + \dots + 2 < 5n. \quad (2)$$

Further, we have to add the time needed to inform all the nodes in the tree about the algorithm termination. Since this task requires at most $3(n-1)$ messages (two COMPUTE and one DIFFUSE messages per MST edge), the construction phase takes at most $8n - 3$ time units to complete.

Until now we have assumed that all the nodes enter in the construction phase at the same time. If this is not the case, then we can use a special message (similar to START_MSG) after the initialization phase. Since the longest propagation path in G is n , in this case, the construction phase will take at most $9n - 3$ time-units. \square

Finally, from Lemma 2 and Lemma 5 we have directly the following result:

Theorem 2 *Given a graph G with n nodes the algorithm for constructing MST takes $O(n)$.*

4 Experimental Results

We have implemented and tested the MST algorithm in the context of the XTV (X Teleconferencing and Viewing) system [2]. As part of this system we have used the spanning tree topology to interconnect a set of information servers that provide the backbone for distributing and advertising conference related information.

The tests and the time measurements for our current implementation were conducted on a collection of twenty workstations DEC/5000 and DEC/3100. All the workstations were connected on the same LAN, and all of them ran the Ultrix 4.2A operating system.

We checked the correctness of our implementation by running tests ranging from three to twenty servers, with ten trials for each number of servers, and comparing them with the results of a sequential implementation of Prim's algorithm [7]. To insure that the constructed MSTs were truly random, rather than allowing the vagaries of network RTTs to govern MST construction, we generated pseudo random RTT values. In each of the 180 trials run, the trees constructed by our algorithm and the ones produced by the Prim's algorithm were identical.

For measuring the performance of our MST algorithm we have performed tests ranging from one to twenty servers⁶, with ten trials for each number of servers. In each case we have considered a completely interconnected graph, i.e., the list of neighbors of each server (*adj_list*) consists of all the other servers. In the following, we derive the worst-case time complexity of our algorithm, and then we show how this compares with the actual measurements.

As shown in our time-complexity analysis in Section 3, the initialization phase takes at most $3n - 2$ time-units (see the proof of Lemma 2) and the construction phase takes at most $9n - 3$ time-units (see the proof of Lemma 5), which results in an overall $12n - 5$ worst-case time-complexity for the entire algorithm. Now recall that in our analysis we have assumed that the propagation delay of any message is at most one time-unit. Therefore in this case we choose the time-unit to be the maximum propagation delay, denoted t_{max} , measured in our tests.⁷ Thus, in the worst-case, our algorithm would take at most $(12n - 5) \times t_{max}$ time to construct a MST in a graph with n nodes.

Figure 8 shows the measured times (for each trial) versus the worst case estimated time. As it can be noticed, in all cases the measured times are well under the predicted worst-case times. Although our results are obtained in a LAN, we are expecting that in a wide area network the results to be at least as good since the parameters that are ignored in our simplified model, such as the message processing and the computation time, are even less important in the presence of much larger propagation delays as experienced in a wide area network.

⁶With one server, of course, the server must merely check to discover that it has no neighbors and announce that minimal spanning tree construction is complete. Measuring this process provides only a rather poor benchmark for the speed of the lone machine.

⁷In computing t_{max} we have used the largest RTT obtained during our tests, which occurred in one of the trials with twenty servers.

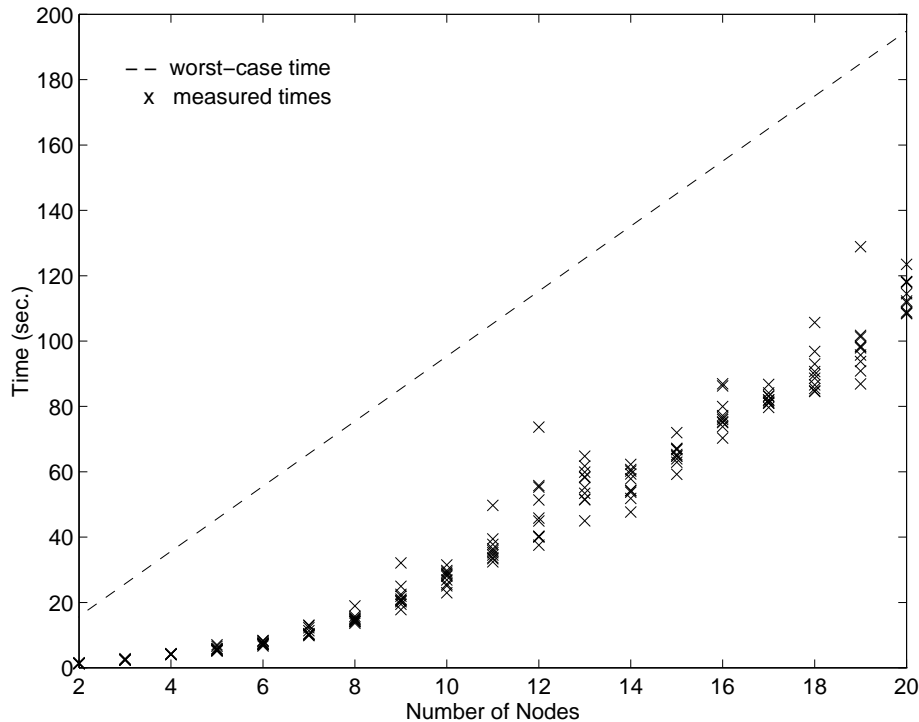


Figure 8: The measured times versus the predicted worst-case time for our MST algorithm. The tests were conducted from two, up to twenty servers; for each number of servers ten trials were performed.

5 Related Work

The previous theoretical solutions for distributed computation of a MST in communication networks are based on the algorithm developed by Gallager, Humblet and Spira [11]. The algorithm exploits the same idea; it starts with all nodes as sub-trees, and iteratively joins them along their minimum-weight edges until the MST is constructed. The algorithm has the message complexity $O(n \log n)$ (which is optimal) and the time complexity $O(n \log n)$. In order to achieve message-optimality some rules are enforced when two trees are joined. Each subtree has associated a level L , such that the number of nodes in the sub-tree is $\geq 2^L$ (a subtree with a single node has level 0). Suppose that a sub-tree T with level L finds that the minimum-weight outgoing edge connects it to the sub-tree T' with level L' . Then, T and T' are joined only if $L \leq L'$, otherwise T waits until T' is big enough. Subsequent refinements of this algorithm have decreased the time complexity to $O(n \log^* n)$ [6] and further to $O(n)$ [3] (which is optimal) but, unfortunately, at the price of increasing its complexity. While the theoretical importance of these algorithms cannot be disputed, their complexity make the practical implementation difficult. As an alternative, we have designed and implemented an algorithm that, although is not optimal in the number of messages, is both simple and time-optimal. Our decision in trading the time complexity over message complexity is justified by the type of applications in the Internet [1, 2, 4] in which the response time tends to be more important for the end users than the number of messages exchanged.

Practical solutions to the multicast problem were based on techniques previously used in distributed routing algorithms [18]. Mainly, current existing solutions for the IP multicast architectures are extensions of distance vector routing (the Distance-Vector Multicast Routing Protocol - DVMRP) [8] and link state algorithms (Link-State Multicast Routing Protocol [9]). A multicast extension to the Open Shortest Path link-state protocol was proposed in [14]. The main disadvantage of these schemes is the scalability problem: computing routing delivery trees is source-based, which requires every router, potentially involved in routing messages for *all* existing groups, to store routing information for *all* potential multicast sources. This induces a scalability factor proportional to the number of (source, group) pairs. In the conditions of a wide area environment, dynamically computing source-based trees becomes a heavy computational task for the routing nodes.

Using a single *multicast delivery tree* for each group connecting the nodes belonging to the same multicast group instead of source-based trees might be seen as a possible solution to the multicast routing in IP networks. In [17] a distributed algorithm to build a spanning tree in a wide area network is presented. Since the algorithm requires that every node to store only a limited amount of information (mainly, the information about the incident links), that is independent of the size of the network, the communication topology is much more scalable. On the other hand, it is no longer homogeneous, namely, the node with the smallest identifier (that becomes the root of the tree) has special functionalities. This has negative impact in the presence of failures, i.e., if the root fails then a high overhead is payed to select a new root. Moreover, depending on the position of the root, the algorithm may generate inefficient communication topologies. For example, consider a network consisting of three nodes A, B, C with the average communication delays: $\tau(A, B) = 500 \text{ ms}$, $\tau(B, C) = 500 \text{ ms}$ and $\tau(C, A) = 2000 \text{ ms}$. If A is the root (has the smallest identifier) then the resulting spanning tree consists of edges (A, B) and (A, C) and therefore a message from B to C takes 2500 ms , while in a tree consisting of edges (A, B) and (B, C) any message takes at most 1000 ms between any pair of nodes.

Recently, a core based tree (CBT) solution was proposed in [4] as an alternative to DVMRP and link-state multicasting architectures, which are cited as suffering not only from poor scalability but also from unicast routing algorithms dependence while being costly in terms of resources consumed at the routing nodes. Here, the scalability problems are eliminated by storing at the routers only *per group* instead of *per source* information. However, as before new problems are generated by the existence of a singularity (the core of the routing tree) subject to failures and by the incurred overhead of dynamically re-selecting and re-placing the core within its tree in order to maintain optimal delivery paths.

6 Conclusions

In this paper, we have presented a fully distributed algorithm to compute a minimum spanning tree in a generic communication network. The underlying communication network is modeled as

a connected, undirected graph in which nodes represent the *host* computers and edges represent the *virtual* interconnections between nodes. In addition, every edge is weighted by the average transmission delay between its end-points. Although this is a simple metric the algorithm can be easily generalized to use more complex ones. We have also proved that our algorithm is time-optimal, i.e., it takes $O(n)$ to compute the MST of a graph with n nodes. To prove the viability of our ideas we have implemented the algorithm and tested it in the context of XTV conferencing system [2].

The main advantage of using a single tree per group for multicast delivery is that it offers a better scalability since a node needs to store only the information associated to its neighbors and not all routing information for *all* potential multicast sources. Moreover, maintaining a spanning tree, on the average, incurs much less overhead when the communication graph changes incrementally (e.g., a node or a link is added/removed). This is because the local changes are usually reflected only among the neighbor nodes and not among all the nodes in the graph (e.g., when a new node is added to the graph only the nodes incident to the modified edges need to update their information). By comparison, if every node would store global information about the communication topology as in link-state routing protocols, then any change will require that every node in the graph to update its information.

The algorithm might be extended in several ways. First, a recovery procedure to handle node and/or link failures must be developed. A simple solution would be to use a variation of the tree construction procedure to re-connect the disjoint trees generated by the failure.

Another problem that should be addressed consists of potentially large variations in network performances due to both transient (e.g., congestions) and persistent factors (e.g., changes in the network topology). Since our algorithm builds the MST based only on data obtained in the initialization phase, the communication tree performances may severely degrade in time. A solution to this problem would be to periodically recompute the MST and to actually use it whenever the gain in performances offsets the overhead introduced by the remapping procedure (i.e., the change of the current communication tree with the new one). An alternative would be to recompute the MST whenever the communication delay between two endpoints increases above a certain threshold.

References

- [1] H. Abdel-Wahab, S-U. Guan, and J. Nievergelt, "Shared Workspaces for Group Collaboration: An Experiment using Inte rnet and UNIX Interprocess Communications", *IEEE Communications Magazine*, Vol. 26, No. 11, November 1988, pp. 10-16.
- [2] H. Abdel-Wahab, I. Stoica and F. Sultan, "The Design and Implementation of an Internet Conference Information System", *Proceedings of the Joint Conference on Information Sciences*, Pinehurst, NC, pp 174-177, November 1994.

- [3] B. Awerbuch, "Optimal Distributed Algorithms for Minimum Spanning Trees", *Proc. ACM Symposium on Theory of Computing*, May 1987, pp. 230–240.
- [4] T. Ballardie, P. Francis and J. Crowcroft, "Core Based Trees (CBT) - An Architecture for Scalable Inter-Domain Multicast Routing", *SIGCOMM '93 Proceedings*, pp 85–94.
- [5] B. Bollobas, "Graph Theory An Introductory Course", Springer Verlag, 1979.
- [6] F. Chin and H. F. Ting, "An Almost Linear Time and $O(n \log n + e)$ Messages Distributed Algorithm for Minimum-Weight Spanning Trees", *Proc. 26th Symposium on Foundations on Computer Science*, Portland , October 1985, pp. 257–266.
- [7] T. H. Cormen, C. E. Leiserson and R. L. Rivest, "Introduction to Algorithms", MIT Press 1992.
- [8] S. E. Deering, "Multicast Routing in Internetworks and Extended LANs", *ACM Trans. on Computer Systems* vol. 8, no. 2, May 1990, pp 85–110.
- [9] S. E. Deering, "Multicast Routing in a Datagram Internetworking", Ph.D Thesis, Stanford, 1991.
- [10] E. W. Dijkstra and C. S. Scholten, "Termination Detection for Diffusing Computations", *Inf. Proc. Letters* 11, Aug. 1980, pp 1–4.
- [11] R. G. Gallager, P.A. Humblet and P. M. Spira, "A Distributed Algorithm for Minimum-Weight Spanning Trees", *ACM Trans. on Programming Languages and Systems*, Vol. 5, No. 1, January 1983, pp 66–77.
- [12] R. M. Karp, "Reducibility among combinatorial problems", *Complexity of Communications*, Plenum Press, New York 1972, pp 85–103.
- [13] M. R. Macedonia and D. P. Brutzman, "MBone Provides Video and Audio Across the Internet", *IEEE Computer* April 1994, pp 30–35.
- [14] J. Moy, "Multicast Extensions to OSPF", *IETF Draft*, July 1993.
- [15] C. A. Noronha Jr. and F.A. Tobagi, "Optimum Routing of Multicast Streams", *IEEE 13th Annual Joining Conference of Computer and Communications, Proceeding vol. 3* 1994, pp 865–873.
- [16] C. A. Noronha Jr. and F.A. Tobagi, "Optimum Routing of Multicast Streams", *IEEE 13th Annual Joining Conference of Computer and Communications, Proceeding vol. 3* 1994, pp 865–873.
- [17] R. Perlman, "An Algorithm for Distributed Computation of a Spanning Tree in an Extended LAN", *Proc. of the 9th Data Communication Symposium*, Sept. 1985, pp. 44-53.
- [18] R. Perlman, "Interconnections", *Addison Wesley* 1992.