The Ordered Core Based Tree Protocol

Clay Shields J. J. Garcia-Luna-Aceves
{clay, jj}@cse.ucsc.edu
Department of Computer Engineering
University of California—Santa Cruz
Santa Cruz, CA 95064

Abstract

This paper presents a new protocol, the Ordered Core Based Tree (OCBT) protocol, which remedies several shortcomings of the Core Based Tree (CBT) multicast protocol. We show that the CBT protocol can form loops during periods of routing instability, and that it can consistently fail to build a connected multicast tree, even when the underlying routing is stable. The OCBT protocol provably eliminates these deficiencies and reduces the latency of tree repair following a link or core failure. OCBT also improves scalability by allowing flexible placement of the cores that serve as points of connection to a multicast tree. Simulation results show that the amount of control traffic in OCBT is comparable to that in CBT.

1 Introduction

Multicast routing protocols build routing trees for the dissemination of messages to a select group of other stations. In some protocols, such as the Distance Vector Multicast Routing Protocol (DVMRP) [1] and the Protocol Independent Multicast-Dense Mode (PIM-DM) protocol [2] the receiving group is assumed to be fairly dense and the sender initiates the multicast assuming all routers in the network are interested in receiving the multicast. If any receiver does not wish to receive the multicast, it must take explicit action and send a message called a *prune* to remove itself from the tree. These types of protocols are termed *sender initiated* as the receivers are not required to take any action to receive the multicast. In each of these protocols the routing tree is formed along the shortest path between each sender and receiver. The overhead at a router $O(n \cdot s)$, where n is the number of multicast groups and s is the number of sources in the group.

In both the Core Based Tree (CBT) multicast protocol [3] and in the Protocol Independent Multicast-Sparse Mode (PIM-SM) protocol [4] [5], a single *shared tree* is created for all sender and receivers in the group, and receivers initiate their own connection to the tree. In each of these *receiver initiated* protocols a well known router exists that accepts connection requests from other routers. This router is known as the *rendezvous point* in PIM; in CBT it is called a *core*. The returning acknowledgment builds a branch of the tree back to the initiator along the reverse path of the connection request. Instead of forwarding each packet on a per-group per-source basis, each data packet is instead forwarded over every on-tree link for that group except the one on which it was received. Accordingly, the router does not have to maintain information about each source for each group and has instead a single entry for each group. The router overhead is therefore $O(n)$, giving the shared tree approach superior scalability. However, because each packet no longer travels over its shortest path to each receiver, shared trees incur longer average delay in the delivery of a data packet.

This work was supported in part by the Office of Naval Research (ONR) under Grant No. N00014-94-1-0688 and by the Defense Advanced Research Projects Agency (DARPA) under Grant No. F19628-96-C-0038.

During times of underlying unicast instability CBT can form loops. Loops in a shared multicast tree are disastrous. When a data packet enters a loop, it circulates the loop endlessly until its time-to-live expires; as a circulating data packet passes through a router that has an off-loop branch, the packet gets forwarded down that branch. This leads to multiple transmissions of each packet in the loop to the rest of the tree. As more traffic finds its way into the loop this situation gets worse, as more and more off-loop transmissions occur. Eventually, the loop can start forwarding so many packets to the rest of the tree that all links on the tree become saturated. We present a new protocol for the construction of shared multicast trees that eliminates this looping problem and other problems that can keep a multicast tree from forming in CBT. We call this protocol the Ordered Core Based Tree protocol.

The next section describes the ways in which CBT can fail. Section 3 describes and specifies the OCBT protocol. Section 4 provides an example of how OCBT handles link failures and is followed in Section 5 with simulation results showing the performance of OCBT and CBT, based on the CBT specifications of April 1996 [6]. We discuss some aspects of core placement in Section 6 before presenting our conclusions.

2 Looping in CBT

CBT [3] [7] [8] [6] forms a *backbone* within a connected group of nodes called cores. The backbone is formed by selecting one router, called the *primary core*, to serve as a connection point for the other cores, called *secondary cores*. Secondary cores remain disconnected from the primary core until they are required to join the multicast group. A router wishing to participate in the multicast session sends a *join-request* towards the closest core. This request travels hop-by-hop on the shortest path to the core, forcing other off-tree nodes to join the branch that the router is forming. When the join-request reaches a core or an on-tree node, a *join-acknowledgment* is sent back along the reverse path, forming a new branch from the tree to the requesting router. If the core that is reached is a secondary core and is off-tree, it connects to a primary core using the same process. Once the tree is constructed, data packets flow from any source to its parent and children. Each parent node forwards the packet to all children other than the sender and to its parent until the data packet reaches the backbone. Each packet is then sent along the backbone and down all other branches, ensuring that all group members receive it.

In the event of a link or node failure, the child node that detects the failure follows a particular strategy in order to reconnect to the tree. If that node's next hop to the nearest core is through one of its immediate children, it sends a message, called a *flush message*, to its children. The flush message travels down the tree, forwarded from parent to child, removing the connection between the parent and child. This message tears down the tree to the individual receivers, which then attempt to reconnect along their best path to a core. If

Report Documentation Page		*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **1997**	2. REPORT TYPE	3. DATES COVERED **00-00-1997 to 00-00-1997**
4. TITLE AND SUBTITLE **The Ordered Core Based Tree Protocol**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of California at Santa Cruz,Department of Computer Engineering,Santa Cruz,CA,95064**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**		
13. SUPPLEMENTARY NOTES		
14. ABSTRACT		
15. SUBJECT TERMS		

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**8**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

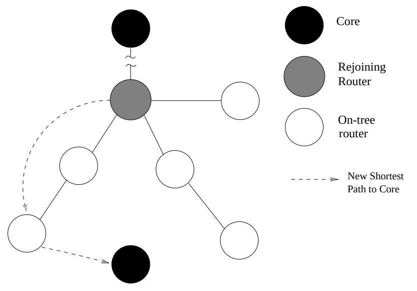

Figure 1: Looping in a disconnected subtree

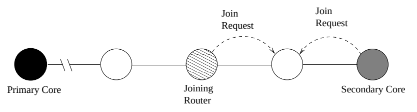

Figure 2: Deadlock or loop formation in tree formation

the next hop to the core is not through a child, the detecting node attempts to reconnect itself by sending a rejoin-request towards the nearest core and does not send the flush message to its children. When the request reaches an on-tree node, that node returns a join acknowledgment that rebuilds the branch down to the sending node. It also sends the rejoin-request to its parent for forwarding to the primary core. The forwarding of the rejoin-request back up the constructed tree is a mechanism used to detect loops that may have formed. If a node receives a rejoin-request that it originated, then a loop has formed. The node detecting the loop removes the link to its parent by sending a message called a *quit-request* and again attempts to rejoin. Otherwise, if the rejoin-request is received at the primary core, that core sends a unicast acknowledgment to the originator of the rejoin request to verify the absence of a loop. This unicast message is needed because if a loop had formed and the rejoin-request was lost before it was returned to the originator, then the loop would not have been detected. However, if the originator never receives the ack, it can assume that a loop has formed, quit from its parent by sending a quit message, and attempt to rejoin again.

Surprisingly, there have been no prior attempts to show that CBT is correct, that is, that CBT creates multicast trees in finite time and that it does not form loops. In fact, CBT does not always form a tree. Part of the tree can remain disconnected when a router seeks to rejoin the multicast tree in response to a failure in the link to its parent. The router detecting the link failure attempts to maintain the sub-tree below it while rejoining the rest of the tree by sending a rejoin-request towards the core. If the path to the core is through an immediate child, the sub-tree is destroyed by transmission of a flush message. Otherwise, the rejoining router sends a rejoin-request towards the core. If this request reaches a descendent in the sub-tree, it is acknowledged and a link forms between the descendent and the rejoining router, completing a loop. Figure 2 shows the topography of a CBT sub-tree subject to this transient looping. The grey node is attempting to rejoin along a newly formed shortest path to the next reachable core. Because its path passes through a descendent child, a loop is formed. This type of loop can occur even when the underlying routing algorithm, which a CBT node uses to determine the path to the core, does not contain loops and is said to be *stable*.

As a loop detection mechanism, the descendent node forwards the rejoin request to its parent, and this message is passed up tree until it reaches the originating router, at which point the loop is detected and action taken to correct it. According to the CBT protocol specification of April 1996 [6], the rejoining router simply sends a quit request to its parent to remove the loop. This correction mechanism can fail, however, as the rejoining router takes no action to destroy its sub-tree and instead attempts to rejoin again, possibly along the same path forming the same loop. Each time the router reconnects along the same path, the same loop forms. This continual looping denies multicast service to the disconnected sub-tree, but it can be stopped if upon detecting a loop, the rejoining router is allowed to flush the tree by forwarding a flush message to all of its children, after which each receiver or sender on the sub-tree connects directly to the core along the shortest path.

If the rejoining router is a secondary core that must reconnect to the primary core, then flushing the tree does not always solve the looping problem. In this case, flushing the tree can initiate a race condition in which local routers attempt to join the secondary core as it attempts to join the primary core. Upon receiving a flush message, routers with members of their subnets desiring the multicast immediately send a join-request on a hop-by-hop basis towards the closest core. That could be a secondary core which is trying to connect to the primary core. If a router that lies on the path to the primary core has attempted to join the secondary core, then it is possible that by the time the join request from the secondary core, which is destined for the primary core, reaches the router, the router will be awaiting an acknowledgment to its own join-request. In this join-pending state the router will accept the local core's request, as illustrated in Figure 2. This will lead to a temporary deadlock, until the appropriate timeouts occur. If these timeouts occur close together and there is no mechanism for selecting an alternate primary core, or if the receiver group near the disconnected secondary core is dense so that each path to the primary core is blocked, this race condition can occur many times, leading to a long latency in reconnecting the sub-tree, if the sub-tree is able to connect at all. A solution to this would be to force routers receiving a flush message to back off for some period of time before attempting to rejoin. While this would prevent the routers from winning the race, it would also lead to long latency times for routers attempting to rejoin the multicast tree.

The same situation that prevents proper reconstruction of the tree following a link failure can also prevent initial construction of the tree, and it can form a loop in a disconnected sub-tree. If a secondary core receives a join-request from a router that lies on the path from the secondary core to the primary core, the secondary core will be unable to form a link to the primary core as all join-requests the secondary core sends will be stopped at the first hop towards the joining router. In Figure 2, this occurs at the white colored router between the joining router and secondary core. In this case the race is always won by the joining router, as it has a head start in sending its join request. As the secondary core sends a join-acknowledgment to the router before attempting to connect to the primary core, the nodes on the path back to the joining router will consider themselves to be on-tree and will acknowledge the secondary core's join-request if the acknowledgment arrives before the secondary core's join-request. A loop, which is undetected by CBT, will then be formed between the router which is the first hop to the joining router and the secondary core. Notice that this loop does not form when secondary core is attempting to reconnect; in the case in which reconnection is occurring the forwarding of the rejoin-request back to the secondary router removes the loop, though the

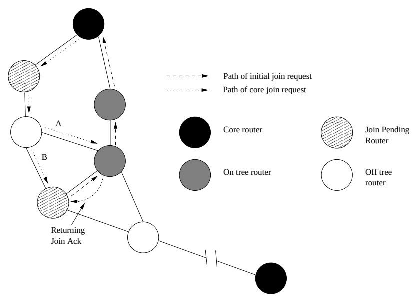

Figure 3: Undetected loop during tree construction

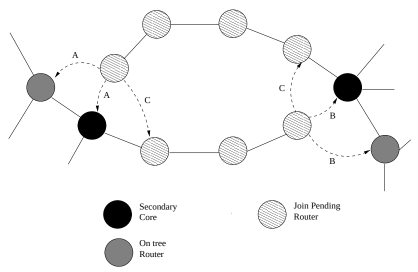

Figure 4: Permanent loops in the core backbone

secondary core will still be unable to connect to the primary core. If the secondary cores do not send a join acknowledgment before sending a join-request, then deadlock can occur as described above.

If the network is unstable during construction, the secondary core's attempt to join the multicast tree can again lead to undetected loops in the disconnected sub-tree. Assume that a router sends a join-request to a secondary core, which is currently off-tree. When the join-request reaches the core, the core acknowledges it and attempts to join the primary core by sending a join-request of its own. If this request travels a different path due to unicast routing instability and traverses a branch of its sub-tree, the join-request will be accepted and acked as shown in Figure 3. The transmission of the ack will occur immediately if the receiving node is on the branch, indicated by path A, or as soon as the ack traveling from the core reaches that point on the branch, indicated by path B. This ack will travel back to the core and form a loop that will not be detected; traffic will circulate within the loop endlessly, dumping repeat copies of data packets down each other off-loop branch as it goes by. Again, this type of loop can only be formed during the initial build of the tree; if the core is attempting to reconnect and uses a rejoin-request instead of a join-request, the loop detection mechanism will detect the loop when the rejoin is forwarded to the core.

There is one similar undetected loop that can form when the tree is constructed during times of network instability or when secondary cores are attempting to contact different primary cores. In this situation it is possible that the primary core is thought to be unreachable by part of the secondary core group without that information having been disseminated through the rest of the group. This inconsistency can occur, due to the mechanism causing deadlock described above, if some secondary core has been unable to reach the primary core because its children blocked the connection and it is now attempting to reach an alternate primary core. If the branches being formed by two secondary cores cross, either due to differences in the destination of the join-request or because of looping in the unicast routing, a loop can be formed. If the loop is formed during the initial construction of the tree, it will be undetected and will not be removed.

If two secondary cores are attempting to form the backbone and their branches meet in a way that forms a loop, one of two things will happen. In the best case, join-pending nodes on each branch receive the request of the other. This is illustrated in Figure 4, where each join-pending node chooses the hop labeled C. No loop will be formed as no acks will be sent; instead, each branch will wait for an ack until they time out.

In the second case, one or both of the forming branches will meet the other at a core or an on-tree node that is a descendent of the core. The core or on-tree router that receives the request will acknowledge it. The ack will travel back along the reverse path forming the branch, possibly getting forwarded back down the other branch that was forming as well. In this case, a loop has been formed in the backbone that will not be detected. Any traffic entering the loop will circle it endlessly, and each time it reaches a router with an interface leading out of the loop, and additional copy will travel down the tree to all receivers. Additional traffic flowing into the loop only serves to exacerbate the situation, resulting in a denial of service as the tree is flooded with the same packets repeatedly. Figure 4 shows how the loop will form if either of any of the next hop choices labeled A or B are taken.

3 The OCBT Protocol

CBT builds a multicast tree from a single level of secondary cores which join at a single primary core. Looping and disconnected sub-trees occur in CBT because the protocol does not enforce any ordering in the way in which nodes and cores attempt to join the tree. In contrast, OCBT maintains a *logical level* for each node and core. The logical level is a label indicating the cores place in the hierarchy of cores. The cores' logical levels are fixed when the core is selected; the nodes levels are not fixed but are assigned when the node joins the tree. Any node or cores level is always less than or equal to the level of its parent; OCBT uses this property to guarantee that no transient or permanent loops ever form in the structure of the tree and that the protocol is safe and live even when routing-table loops occur in the underlying routing protocols. OCBT has been shown to be free of loops at every instant and to be safe and live [9]. OCBT also reduces control traffic following a link failure, allows for flexible core placement, and does this without increasing the complexity of the protocol.

When a router has a member wishing to receive the multicast session, it locates the nearest core and sends a join-request towards that core. Join-requests force any off-tree routers they reach on their path to the core to forward the request and attempt to join the tree. In OCBT, join-requests also carry a field which contains the level a node must have to safely acknowledge the request. Join-requests from an off-tree router carry a level of zero to indicate that any on-tree node or core can safely acknowledge the request. If a node receives a join-request carrying a level higher than its level, it quits from its parent and joins the branch that the join-request is forming. In this way, OCBT forces lower-level branches to break to allow the construction of higher-level branches. This prevents the cases in CBT in which a node or core attempting to rejoin following a link failure is unable to connect to a core because it is blocked by its sub-tree, preventing that sub-tree from joining the main multicast tree.

OCBT limits control messages to within a particular logical level and distributes the processing of control messages over a larger number of cores. When a link fails, flush messages travel down-tree only as far as the next lower level of cores; join-requests need only travel as far as the next higher level of cores. This results in less traffic following a link failure than in CBT, in which flush messages from near a core or rejoin messages originating far from the core have to travel relatively long distances. More recent specifications for CBT [6] have a single primary core that forms a point of connection for secondary cores that stay off-tree until required to join. This single primary core is a limiting factor to the scalability of CBT, as it must receive and respond to all passive join-requests from the entire multicast tree. OCBT has no similar single point of traffic concentration, as cores need only respond to traffic within its logical level.

Other differences between CBT and OCBT include changes in the mechanism by which nodes destroy the connection formed with their parent. OCBT replaces the *quit request* of CBT with a *quit notice*, and in OCBT nodes sending the quit request do not wait for an acknowledgment before leaving the tree. In contrast, under CBT, nodes must wait for an acknowledgment from the parent before leaving the tree. OCBT uses a keep-alive mechanism to detect lost quit-notices and flush messages instead of using explicit acknowledgments. A *parent-assert* message is included in OCBT to insure that consistent state information is maintained between nodes. A parent keeps track of reception of keep-alive packets from its children. In the event that the parent does not receive a keep-alive from a child in a set period of time, it sends a parent assert message to ascertain if the child still is its child; if no reply or a negative reply to a parent assert is received, the child is assumed to have quit. This guarantees eventual consistent information about the state of the link between child and parent, even if messages are lost. Because no node accepts or forwards an on-tree data packet from an off-tree link, no data packets are received twice, even if a quit-notice or flush message is lost.

OCBT is quite similar in complexity to the original CBT. OCBT takes $O(n)$ to create a spanning tree, where n is the number of links in the spanning tree and is dependent on the network, core placement and multicast group members. The load on the routers is only marginally increased. In addition to the state variable required for CBT, each on-tree router in OCBT is additionally required to track its level and to maintain level information for each of its children, as well as a marker as to whether that child has transmitted a keep-alive packet recently.

OCBT's specification is shown in Figure 5. Function names are in **bold**. A call to another function or the name of a particular type of message is capitalized. Parameters that are part of a received message are in *italics*. Names of the variables maintained within the node are plain, lower case.

Each of the cores and routers maintains variables representing the state of the node in regard to OCBT's operation. Each node has an entry for its OCBT state (on-tree or off-tree or join-pending, and core or non-core), level, parent, the core it last attempted to reach, and a list maintaining the list of the node's children and their level. Core nodes also have one additional state variable, which is the logical level of their parent. This entry is used to track the core state in case it is coerced to a higher level; if for some reason it receives a flush message from its parent, it can flush all children of level greater than the original core level and return to that level.

Examination of OCBT's specifications reveals that descriptions of some called functions are missing. In particular, **Next Hop**, **Find Core**, **Subnet Member** and **Send Message** were omitted for brevity, but are explained below.

Subnet Member determines whether the router has some member on its local network wishing to receive the multicast; if it does, this function returns true.

Send Message transmits a message to the designated recipient that includes the information specified; if the message being sent is a join-request, **Send Message** also starts the timeout timer. Receipt of an appropriate acknowledgment cancels the timer.

Next Hop examines the unicast routing table and returns the neighbor node on the next hop to take towards a given destination.

Find Core returns the nearest core of a specified level; if level 0 is specified, it returns the closest core of any level. **Find Core** was omitted as the actual OCBT code depends on the means used to distribute core information. If some means of scoping is desired, **Find Core** may not return the closest core, but instead one that lies within the scoped area. **Find Core** changes the node variable `core`; each time **Find Core** is called, `core` is updated to whatever it returns. In addition to locating cores, **Find Core** also detects partitions in the network when higher-level cores are unreachable and instigates a partition-recovery mechanism. In order to do this, it maintains a list of cores that have been contacted but failed to respond; this list is cleared when the node is joining and receives an ack.

4 Tree Maintenance in OCBT

OCBT builds a distribution tree in which each member has a logical level equal to or less than its parent. The logical level changes only at a core or a *graft*. Grafts occur where a lower-level branch is broken to make way for a higher level branch to form, and the lower level branch is maintained below the break. Figure 6 shows the structure of an OCBT tree. The large nodes are cores and show their levels. The smaller black nodes are on-tree nodes and have the link to their parent labeled with their level. The striped node is a graft node which formed when the $(n+1)$-level branch broke to allow the $(n+2)$-level branch to connect to the $(n+2)$ core.

When a link failure requires recovery of the tree, cores and grafts respond in different manners. A core attempts to reconnect for its children; a graft flushes the tree below it and expects a core or receiver below it to attempt reconnection. Figure 7 illustrates this by showing the state of the tree after a link failure. Following the link failure, the $(n+1)$-level core and the leftmost level-n core would each attempt to reconnect to their higher-level core. If the network remained partitioned and the $(n+2)$-level core was unreachable, the multicast tree would form up to the $(n+1)$-level core, which would then wait until the partition was corrected to rejoin the multicast tree.

5 Simulation Results

To examine the performance of CBT and OCBT in a realistic manner, we created a simulation of each protocol using a simulation package* that supports protocol layering. These simulations ran on top of a unicast routing layer that implemented the distributed

*The protocols presented in this paper were simulated using the C++ Protocol Toolkit (CPT) by Rooftop Communications Corp. of Los Altos, CA.

```
Add Child (child,level)
  Add Child to List (child, level)
  Send Message (Join Ack, child, level)

Break Branch (source, message level,
              core, originator)
  Send Message (Quit-Notice, parent)
  if (state = On-Tree Core) or
  (state = Join-Pending Core)
    parent level = message level
  parent = Next Hop (core)
  if (On Child List (parent))
    Remove Child from List (parent)
  Add Child (source, message level)
  send message (Join-Request, parent,
          message level, core, originator)
  if (state = On-Tree Core)
    state = Join-Pending Core
  else
    state = Join-Pending

Forward Message (type, source)
  for each child
    if (child != source)
      Send Message (type, child)
  if (parent != source)
    Send Message (type, parent)

Join Tree (level)
  if state = Join-Pending Core
    parent = Next Hop (Find Core(level + 1))
    Send Message (Join-Request, parent,
              level + 1, core)
  else /* level = 0 */
    parent = Next Hop (Find Core(level))
    Send Message (Join-Request, parent,
              level, core)
    state = Join-Pending

Multicast Message (type, level)
  for each child on list
    if (level = 0) or (level < child level )
      Send Message (type, child, level)

Quit Tree ()
  parent = null
  if (state = On-Tree Core)
    or (state = Join-Pending Core)
    parent level = core level
    state = off-tree core
  else
    state = off-tree
    level = 0
  halt /* do not return */

Remove Children (level)
  for each child on list
    if (level = 0) or (level < child level )
      Remove Child from List (child)
  if (child list = null)
    and not (Subnet Receiver)
    Quit Tree
  else
    return to calling function
```

```
Remove Child (child)
  Remove Child from List (child)
  if (child list = null) and
    not (Subnet Receiver)
    Send Message (Quit-Notice, parent)
    Quit Tree
  else
    return to calling function

Send Data (source, data)
  if (source = parent) or
    (On Child List (source))
      Forward Message (data, source)
    else
      drop the packet and
      do not forward to subnet

Join-Pending or
On-Tree Router (message type, message level,
               source, core, originator)
  case (message type)
    Join-Request
      if (on child list (source))
        Remove Child from List (source)
        if (message level > level)
          Break Branch (message level,
                        core, originator)
        else
          if (state = On-Tree Router)
            Add Child (level)
          else
            Add Child to List (source)
    Quit-Notice
      if (on child list (source))
        Remove Child (source)
    Flush Message
      if (source = parent)
        Forward Message (Flush Message, source)
        Remove Children (0)
        /* only reached if above function returns */
        level = 0
        Join Tree (level)
    Join Ack
      if (state = Join-Pending Router)
        if (source = parent)
        and (message level >= level)
          level =message level
          Forward Message (Join Ack, level,source)
    Data
      Send Data (data, source)

Off-Tree Router (message type, message level,
               source, core, originator)
  case (message type)
    Join-Request
      parent = Next Hop (core)
      level = message level
      Send Message (Join-Request, parent,
                    level,core)
      state = Join-Pending
```

```
Join-Pending Core or
On-Tree Core (message type, message level,
              source,core,originator)
  case (message type)
    Join-Request
      if (on child list (source))
      /* previous quit-notice was lost */
        Remove Child from List(source)
      if (message level <= level)
        Add Child (source, level)
      else
        if (message level > parent level)
          Break Branch (message level,
                        core, originator)
        else
          if (On-Tree Core)
            Add Child (source, parent level)
          else
            if (originator = self)
              /*message looped - unicast instability */
              Send Message (Quit-Notice, parent)
              Send Message (Flush Message,source)
              parent level = level + 1
              parent = Next Hop (Find Core(level + 1))
              Send Message (Join-Request, parent,
                            level + 1, core)
            else
              Add Child to List (source, message level)
    Quit-Notice
      if (on child list (source))
        Remove Child (source)
    Flush Message
      if (source = parent)
        Multicast Message (flush message, level)
        Remove Children (level)
        /* only reached if above function returns */
        state = Join-Pending Core
        Join Tree (level)
    Join Ack
      if (Join-Pending Core)
        if (source = parent)
        and (message level > level)
          parent level = message level
          foreach child on list
            if (child level > core level) and
            (child level <= parent level)
              send message (Join Ack, child, parent level)
          state = On-Tree Core
    Data
      if (On Tree Core)
        Send Data (data, source)

Off-Tree Core (message type, message level,
              source, core, originator)
              case (message type)
    Join-Request
      if (message level <= level)
        Add Child (source,level)
        parent = Next Hop (Find Core(level + 1))
        parent level = level +1
        Send Message (Join-Request, parent,
                      level + 1, core)
      else
        Add Child to List (source, message level)
        parent = Next Hop (core)
        parent level = message level
        Send Message (Join-Request, parent,
                      message level, core)
      state = Join-Pending Core
```

```
On Time Out
  case (state)
    Join-Pending Core
      if (parent level > core level + 1)
        for each child on list
          if (child level > level)
            Remove Child from List (child)
      if (child list != null) or (Subnet Member)
        parent = Next Hop (Find Core(level + 1))
        parent level = level + 1
        Send Message (Join-Request, parent,
                      level, core)
      else
        parent = null
        parent level = level
        state = Off-Tree Core

    Join-Pending Router
      for each child on list
        Remove Child from List (child)
      parent = null
      level = 0
      if (Subnet Member)
        Join Tree (0)
      else
        state = Off-Tree Router

On Parent Link Failure
  case (state)
    On-Tree Core or
    Join-Pending Core
      Multicast Message (Flush Message, level)
      for each child
        if (child level > level)
          Remove Child from List (child)
      if (Subnet Member) or (child list != null)
        state = Join-Pending Core
        Join Tree (level)
      else
        parent = null
        parent level = level
        state = Off-Tree Core

    On-Tree Router or
    Join-Pending Router
      Forward Message (Flush Message, parent)
      for each child
        Remove Child from List (child)
      level = 0
      if (Subnet Member)
        Join Tree (level)
      else
        parent = null
        state = Off-Tree Router
```

Figure 5: OCBT Protocol Specification

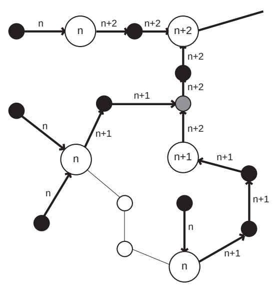

Figure 6: HCBT Tree

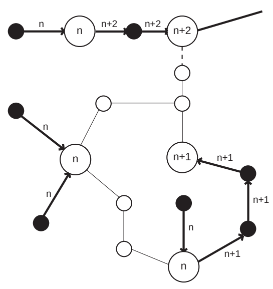

Figure 7: Link Failures

Bellman-Ford algorithm and used routing information from the unicast layer. Using this simulation we measured the end-to-end delay of data packets traversing the tree, the number of messages of each type sent before and after a link failure, and the number of times CBT formed of transient loops requiring explicit action from CBT to remove. In addition, each case in which a CBT sub-tree was un-

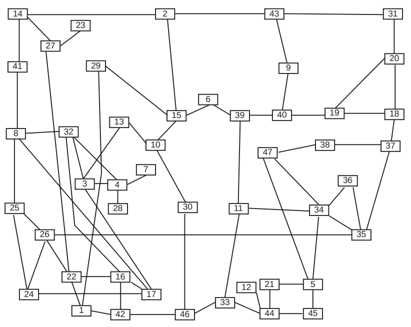

Figure 8: Arpanet Simulation Topography

able to reconnect to the tree, as shown in Figure 2, was recorded. We also recorded the number of times OCBT did not form transient loops when CBT would have.

For our simulations we used the Arpanet topology shown in Figure 8, which contains 47 nodes and 69 edges. We examined the performance of OCBT and CBT under realistic conditions: the links on the network were configured to run at 200 kilobits per second, with a 1 millisecond delay between hops; the unicast routing updates occurred four times as frequently as the CBT and OCBT keep-alive messages. We chose this update period to allow the unicast routing time to disseminate routing information; this was important because one indication of a link failure was a change in the unicast routing table. We selected two receiver groups for the simulation - a dense group consisting of all nodes and a sparse group consisting of 11 widely distributed nodes. The same single source was used with each receiver group.

For each run of the simulation, we chose a particular set of cores using what is probably the same "trivial heuristic" used by Ballardie [3], that is, looking at a picture of the network we picked distributed nodes of relatively high degree to serve as cores. For OCBT, the cores were divided into two logical levels. We constructed the CBT backbone before allowing receivers to connect even though the current protocol specification does not; we did this because of the difficulty CBT has in connecting secondary cores to the primary cores. Building each of the trees for each receiver group, we measured the construction costs in terms of the traffic required. We then sent a stream of data packets from the source to all receivers and recorded the delay each data packet encountered. Finally, we made each link in the network fail individually and measured the number of messages required to reconnect the tree and any loops that were formed.

In our simulation, link failures were detected in two ways. First, failure of a parent or child to respond to a set number of keep-alive messages created the link-down condition. Second, every time a message was sent, the unicast routing was checked to see if the next hop to the destination had changed. Changes in the next hop information reflect a change in the underlying unicast routing that came about as result of a link failure. This allowed the protocol to detect link failures before the set number of keep-alive messages were lost.

The simulations used the 12 different core sets shown in Figure 1, with the results summarized in Table 2. Each run shows the average performance of OCBT and CBT for a sparse and dense receiver group for the selected core set. The delay and the variance of the delay are normalized to the delay and variance of a source based

Run Number	Level 1 Cores	Level 2 Cores
1	3 40 34	15 33 26
2	32 33	40 26
3	40 26 32 33	3 40
4	2 10 46	8 37
5	33	15
6	26 33	40
7	40 26 32 33	15
8	26 44 18	30
9	40 26 32 33	2 10 46
10	37	2 46
11	14 24 31 45	15 30
12	17 44 31	34 32

Table 1: Cores used in simulation

Run #	Build Messages	Repair Messages	Average Delay	Delay Variance	Trans. Loops	Disconn. Subtrees	Loops Prevented
1							
OCBT	71	4.4	1.5	2.5	-	-	0
CBT	70	14.2	1.5	2.7	0	9	-
2							
OCBT	72.5	4.2	1.4	2.0	-	-	0
CBT	68	4.4	1.4	1.9	0	17	-
3							
OCBT	71	4.3	1.3	1.6	-	-	0
CBT	73	4.4	1.9	3.5	0	19	-
4							
OCBT	80	5.5	1.2	1.5	-	-	0
CBT	72	4.5	1.7	3.0	0	18	-
5							
OCBT	72	5.2	1.3	1.6	-	-	0
CBT	70	15.3	1.2	1.6	0	17	-
6							
OCBT	70	4.5	1.5	2.1	-	-	0
CBT	66	10.9	1.5	2.1	1	18	-
7							
OCBT	79.5	4.3	1.6	2.8	-	-	0
CBT	69	12	1.6	2.4	1	13	-
8							
OCBT	80	7.4	1.4	1.8	-	-	0
CBT	69	4.1	1.4	1.8	0	10	-
9							
OCBT	78.5	4.5	1.6	3.0	-	-	4
CBT	73	21.4	1.2	0.5	1	11	-
10							
OCBT	75	6.0	1.7	3.1	-	-	2
CBT	72	4.9	1.2	0.5	0	23	-
11							
OCBT	87.5	5.9	1.2	1.5	-	-	2
CBT	71	4.1	1.5	2.1	0	20	-
12							
OCBT	75	4.8	1.4	1.8	-	-	0
CBT	68	3.6	1.4	1.8	0	12	-
Average							
OCBT	76	5.1	1.43	2.1	-	-	.67
CBT	70	8.7	1.46	2.0	.25	15.6	-
Source	72		1.00	1.00			
.95 C.I.							
OCBT	72.7-79.3	4.5-5.7	1.3-1.5	1.7-2.5	-	-	0.0-1.5
CBT	68.7-71.5	4.9-12.4	1.3-1.6	1.4-2.6	0.0-0.5	12.8-18.4	-

Table 2: OCBT vs. CBT

tree. The source based tree was created using CBT with a single core located at the sender for the same two receiver groups. The delay results were then averaged. The Transient Loops entry for CBT shows the number of transient loops that were able to be corrected. The Disconnected Subtree column shows the number of times a CBT sub-tree was unable to reconnect to the main tree following a link failure. The Loops Prevented entry shows the number of loops caused by instability in the unicast routing that OCBT detected and which would have formed transient loops in CBT.

The results demonstrate that the major advantage of OCBT is its loop freedom and its ability to correctly reconstruct a multicast tree following a link failure. In our simulations, a CBT sub-tree was frequently unable to reconnect to the multicast tree following a link failure as described in section 2. As each set of simulation runs included 138 runs of the CBT protocol, and an average of 15.6 disconnected sub-trees were formed during those runs, we found a disconnection rate of 11.3% under the current protocol specifications [6]. Clearly, a routing protocol that is unable to find a correct path when one exists one time out of nine is hardly suitable for use in a large Internet.

The message count for the CBT protocol was kept artificially low in situations when a sub-tree was unable to reconnect, as our simulation enforced a timeout period for any rejoining node that detected a loop. Had those routers been allowed to attempt to connect as quickly as possible, the total number of messages would have been much higher. In addition, we formed the CBT backbone before the receivers were allowed to join; this also lowered the total message count as it prevented situations in which a secondary core could not connect to the primary core.

On average, OCBT requires some additional work to build the tree, but once it is constructed the traffic required to maintain the tree is reduced. Intuitively, one might expect the OCBT tree to require less traffic to build, as lower-level cores remain in an off-tree state until they receive a join-request. If a particular core never receives a request for the multicast session, it can remain off-tree and no traffic is required to build the tree out to it. However, we found that the OCBT takes slightly more messages to form the multicast tree than that of the version of CBT we tested. This is because many children tried to connect in close succession to lower-level cores that were off-tree. As these lower-level cores sent join acks to the joining nodes before attempting to reach a higher-level core, many links were formed between the core and its new children. The lower-level core was then forced to break some of these existing links to reach the higher-level core. Links formed this way required five messages to form - two for the initial node to core join, two for the link to form from between the lower-level core and the higher-level core, and one quit-notice sent from the child to its former parent as it was coerced to join the higher-level branch that was forming.

OCBT did reliably reform the tree after a link failure with fewer messages than CBT. The branches of the CBT tree can grow fairly long, and messages can be required to traverse the entire length of the branch in the event of a link failure. If the failure is near the bottom of the branch and a rejoin occurs, CBT requires that a passive rejoin be forwarded the length of the branch to the primary core, which then sends a unicast message to the originator acknowledging the passive rejoin-request to ensure that there is no loop formed. As the unicast message does not necessarily traverse the multicast tree on its return to the originator, we did not include it in our message count as it may not contribute to on-tree congestion.

Similarly, if the failure is near the backbone and the branch is flushed, then the flush must travel the length of the branch to the receivers which then send a join-request back to a core, resulting in messages traversing the branch twice. OCBT reduces the traffic requirements in both cases. OCBT does not require that a rejoin-request be forwarded to the highest-level core; instead it only travels as far as the next higher core as required to rejoin the tree. The flush message cannot destroy a branch all the way from the highest-level core down to the receivers as control traffic is limited to a single logical level.

As expected, the multicast trees produced by CBT and OCBT produce more delay in delivering packets than do source-based trees. This can be seen intuitively as the path a packet would take in a core based tree might not be the shortest to each receiver since it must detour to pass through a core. The actual delay from a source to a receiver is dependent on core placement, as no additional delay will be incurred if the core lies on the shortest path. With poor core placement in an OCBT tree, this could be exacerbated as the packet may be routed further off the shortest path to pass through several cores. In our simulation, the delays experienced by data packets in OCBT were on average about 43% greater than the delay experienced by a packet from a source-based tree. Data packets sent over the tree formed by CBT experienced an average delay about 46% greater than the source based tree. Using OCBT, it is possible that this could be reduced by making each source a lower level core. Nearby nodes would then connect directly to the source, while nodes further away would receive the multicast over the shared backbone.

Both CBT and OCBT construct and operate a fixed tree. This has the clear drawback of requiring all data packet transmissions to traverse specific links in the network, regardless of congestion. This can create *hot spots* at cores that must handle an excessive amount of traffic. CBT is more susceptible to hot spots, particularly at the primary core which must receive and reply to each passive rejoin request. Using more cores can alleviate hot spots somewhat, as this spreads the traffic over more cores, though this does not reduce the traffic at CBT's primary core. OCBT is more amenable to use of additional cores, and does not require any single core to answer messages from the entire multicast group. Another partial solution to congestion over fixed links is to allow children to quit from their parents and connect on a shorter path to the core if one becomes available. This in fact was first suggested by Ballardie for CBT [7], but has not yet been included in our simulation. Another improvement to be investigated will be to make each source a local core so that near by nodes can join directly to it, reducing the delay to those nodes.

The slightly increased number of messages required in the construction of the tree is a very small price to pay for OCBT compared to its major advantage: it works correctly. CBT, in contrast, is incorrect and does not always form a complete multicast tree during construction or following a link failure. Permanent, undetected loops can form in CBT that can cause complete saturation of every link on the tree containing the loop. This is clearly an undesirable characteristic of CBT; OCBT suffers from no similar detrimental traits and can be used safely. In addition, in a tree with many link failures, OCBT's reduced repair costs actually makes the amortized cost of construction lower than CBT.

6 Core Placement

There are a number of issues concerning the placement of cores in the network and the distribution of information about the core location. Currently, we assume that some mechanism for distributing core information is universally available and that each router can find the address and level of any core. In reality, this is neither desirable nor possible. A leaf router within the United States has little use for information about local cores in other countries, nor does it have the space to maintain what could be large core lists. Instead, some mechanism for leaf nodes to discover local cores and for lower-level cores to become aware of nearby higher-level cores is needed. This could take the form of a multicast group server able to respond with the identities of local cores, similar to the DNS service.

Alternatively, cores could follow a distributed scheme for disseminating their location and level, broadcasting or flooding their identity and location with an increasing time-to-live over each of their interfaces, or they could join a multicast group that existed solely for the purpose of core location dissemination. Cores need only know the addresses of the same level and next higher-level cores, so some method of limiting the core information that gets distributed is desirable. Multicast distribution schemes could also work well in a situation in which the multicast was being limited to a particular *scope*, that is, limiting the area of the network in which the multicast tree forms. Each level of cores in the scoped area could have its own local multicast address. A scheme similar to this was proposed in HPIM [10]. The issue of core information distribution is an area of future work for OCBT and other protocols based on shared trees.

After the cores are identified and a means of determining the location of nearby cores is established, the issue arises of whether or not to build a backbone of cores prior to allowing any leaf nodes or lower level to connect. In OCBT this is not strictly necessary, although it can help prevent some worst case behavior, in which the highest-level cores are forced to break many existing links if other connections are made before the backbone forms. In CBT it is not necessary unless one wants to be certain that secondary cores can join the tree. In OCBT the backbone is formed by choosing one core of the highest-level to be a connection point for all of the other highest-level cores. This core undergoes a temporary promotion to one level higher then the rest of the highest level cores. The other highest-level cores then join the promoted core.

The core placement in OCBT has an important effect on the performance of the protocol in terms of the amount of control traffic generated and the delay imposed on data packet delivery. This is true in CBT and PIM-SM as well. While determining optimal core placement remains an open problem, there have been suggestions made as to methods of migrating cores to provide better service [11] [10]. We believe that core placement can be made a matter of policy rather than optimality if the scope of the multicast is limited at each level. The flexibility of adding additional cores in OCBT supports this approach. Core placement and migration are important issues for our future work.

7 Conclusions

We have described an ordered extension to CBT, called OCBT, that increases scalability, reduces repair latency, completely eliminates loops, and is provably correct in forming a multicast tree. By distributing cores throughout the network and by maintaining logical level information, OCBT allows for a flexible multicast group in which the core structure does not have to be fixed in advance. The distribution of cores reduces the amount of repair traffic by limiting the distance over which repair messages have to travel to within the logical level.

OCBT eliminates the loops and disconnected sub-trees that occur in the CBT protocol [9]; our simulation results corroborate our verification work. The cost of OCBT is a slight increase in the initial number of messages required to construct the multicast tree. This is somewhat balanced by a reduction in the amount of traffic required to repair the tree following a link failure, and a guarantee that the tree will reform correctly. The increase in tree construction traffic is a result of the mechanism that breaks lower-level tree branches to allow formation of a higher-level branch; in some cases, this mechanism also adversely affects the number of messages it takes to repair a failure in the tree. On average, however, OCBT reconstructs the tree with less traffic than CBT and does so correctly; in all cases the multicast tree will be formed correctly and will reform correctly following a link or node failure.

The delay induced in end-to-end packet delivery by OCBT is comparable to that of CBT: both increase the average delay by about 50% over the delay of a source-based tree. The actual delay incurred is dependent on the location of the cores. It may be possible to reduce the delay in OCBT trees by making each source a local core. Nearby nodes would then be able to connect directly to the source, minimizing their perceived delay, while more remote receivers would connect via the shared tree.

The relative number of messages and delay induced by CBT and OCBT are hardly indicative of the overall performance of each protocol. The Core Based Tree protocol is incorrect; it does not prevent or detect looping nor does it consistently build a correct multicast tree. The correct construction of the multicast tree in all instances and the guarantee of loop freedom in the Ordered Core Based tree protocol make it superior in operation to CBT; it is only an added bonus that it does so with a reduced amount of control traffic. The changes that make OCBT perform correctly and more efficiently than CBT are simple and extensible; work done on the placement of cores and security mechanisms for CBT are applicable to OCBT with little or no modification. The need for a scalable multicast routing protocol in the Internet of the future highlights the importance of a shared tree protocol; OCBT meets that need with correct and efficient performance.

REFERENCES

1. S. E. Deering, *Multicast routing in a datagram internetwork*. PhD thesis, Stanford University, Palo Alto, California, Dec. 1991.
2. D. Estrin, D. Farinacci, V. Jacobson, C. Liu, L. Wei, P. Sharma, and A. Helmy, "Protocol independent multicast-dense mode (PIM-DM): Protocol specification," Internet Draft draft-ietf-idmr-pim-dm-spec-01.txt, U.S.C, L.Bl., CISCO, January 1996.
3. A. Ballardie, P. Francis, and J. Crowcroft, "Core based trees (CBT)," in *Proc.of the ACM SIGCOMM93*, (San Francisco, California), pp. 85–95, ACM, Sept. 1993.
4. S. Deering, D. Estrin, D. Farinacci, M. Handley, A. Helmy, V. Jacobson, C. Liu, P. Sharma, D. Thaler, and L. Wei, "Protocol independent multicast-sparse mode (PIM-SM):protocol specification," Internet Draft draft-ietf-idmr-pim-sm-spec-02.txt, XEROX, USC, CISCO, UCL, LBL, UMICH, May 1996.
5. S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C. Liu, and L. Wei, "An architecture for wide-area multicast routing," in *Proc.of the ACM SIGCOMM94*, (London, UK), pp. 126–135, Sept. 1994.
6. A. Ballardie, S. Reeve, and N. Jain, "Core based trees (CBT) multicast protocol specification," Internet Draft I-D, University College London, April 1996. Work in progress.
7. A. Ballardie, *A New Approach to Multicast Communications in a Datagram Internetwork*. PhD thesis, University College London, University of London, London, U.K., 1995.
8. A. Ballardie, "Core based trees (CBT) multicast architecture," Internet Draft I-D, University College London, February 1996. Work in progress.
9. C. Shields, "Ordered core based trees," Master's thesis, University of California, Santa Cruz, Santa Cruz, California, June 1996.
10. M. Handley, J. Crowcroft, and I. Wakeman, "Hierarchical protocol indepenent multicast (HPIM)." University College London, November 1995.
11. K. Calvert, R. Madhavanand, and E. Zegura, "A comparison of two practical multicast routing schemes," Tech. Rep. GIT-CC-94/25, College of Computing, Georgia Institute of Technology, Atlanta, Georgia 30332-0280, February 1994.