University of California
Santa Cruz

Ordered Core Based Trees

A thesis submitted in partial satisfaction
of the requirements for the degree of

Master of Science

in

Computer Engineering

by

Clay Shields

June 1996

The thesis of Clay Shields is approved:

Prof. J.J. Garcia-Luna Aceves

Prof. Tracy Larrabee

Prof. Darrell Long

Dean of Graduate Studies and Research

Copyright © by

Clay Shields

1996

Contents

Abstract **vii**

Acknowledgments **viii**

1. INTRODUCTION **1**

2. BACKGROUND **4**

2.1 Looping in CBT 5

3. THE OCBT PROTOCOL **11**

3.1 Differences between OCBT and CBT 11

3.2 Core Placement 13

3.3 Tree Construction 14

3.4 Tree Maintenance 18

3.5 Network Partitions 19

3.6 OCBT Specification 21

4. CORRECTNESS OF OCBT **27**

4.1 Connectivity in a Connected Network 27

4.2 Loop Freedom 28

4.3 Connectivity in a Partitioned Network 41

5. PERFORMANCE OF OCBT **44**

5.1 OCBT and CBT 46

5.2 CBT implementations 50

6. CONCLUSIONS **53**

6.1 Future Work 54

References **56**

List of Figures

2.1 Looping in a disconnected CBT subtree. 6

2.2 Deadlock or loop formation in tree formation 7

2.3 Undetected loop during tree construction under routing instability 9

2.4 Permanent loops in the core backbone formed under routing instability . . . 10

3.1 Initial OCBT Tree Building Process 15

3.2 Building the tree from the Lower-Level Cores 16

3.3 Building the tree from the Higher-Level Cores 16

3.4 Build completion and Data Traffic Routing 17

3.5 Link Failures 18

3.6 Routing Loop Unrolling 19

3.7 Common OCBT Functions 23

3.8 OCBT Protocol for Core Nodes 24

3.9 OCBT Protocol for Router Nodes 25

3.10 OCBT Protocol for Timeouts and Parent Link Failures 26

5.1 Arpanet Simulation Topography 45

List of Tables

5.1 Cores used in simulation 47

5.2 OCBT vs. CBT 48

5.3 The performance of CBT implementations 52

Ordered Core Based Trees

Clay Shields

ABSTRACT

This thesis presents a new protocol, the Ordered Core Based Tree (OCBT) protocol, which remedies several shortcomings of the Core Based Tree (CBT) multicast protocol. The CBT protocol can form loops during periods of routing instability, and it can fail to consistently 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.

Keywords: Multicast, Routing, Loop-free

Acknowledgments

I would like to thank my advisor, Dr. J.J. Garcia-Luna Aceves for providing the initial idea that led to this body of work, and for his patience, support and forbearing while the simulation work was almost complete time and time again. Thanks to Dave Beyer for his virtually unlimited support on all my questions about CPT, the simulation tool used to implement the OCBT and CBT simulations. Dr. Tracy Larrabee is responsible for all good grammar contained in this work, that leads me to thank her for this.

I would like to thank my parents, particularly my mom, for their understanding when I chose to return to school instead of getting a "real job" and to reassure them that I'll be out of school and working real soon now, maybe. I certainly also need to acknowledge my friends and housemates who kept me passably sane the past two years.

This work was supported in part by the Office of Naval Research under Contract No. N-00014-92-J-1807.

1. INTRODUCTION

As computer networks become more pervasive and the technology is applied in new and different ways, transmissions that conserve system resources will become increasingly crucial. One common type of network messages are *unicast* transmissions; these are sent from one computer to one other specific computer. There are also *broadcast* transmissions that go from one station to all other stations, though these are more typical of television and radio than of the Internet. Becoming increasingly common are *multicast* transmissions, which fill the gap between unicast and broadcast: multicast allows a message to be transmitted to a select group of other stations. The challenge of multicasting is to do this efficiently.

Currently, if one were to send an e-mail message to a group of friends who use different computers, the message would be sent from the originator one time for each recipient. This is not a problem with e-mail, as the messages are fairly small and the multiple copies that are being transmitted do not overload the outgoing network link. However, if the same originator were trying to send a digital high definition television signal to a large group receiving a pay-per-view program, such duplication of the signal would undoubtedly quickly overload the network. Multicast routing protocols provide ways to route data packets so that only one copy ever traverses any given link. In a way, both unicast and broadcast are special cases of multicasting; the first being a multicast group with a single receiver and the latter transmitting to a multicast group of all possible receivers.

There are a variety of routing protocols that approach the problem in different ways. 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. In these protocols the sender initiates the multicast assuming all routers in the network are interested in receiving the transmission and, initially, the multicast is sent to all receivers. 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 prune messages have a limited lifetime; every so often the prune messages expire

and the multicast goes to all possible recipients, which again have to prune the branch if they still do not wish to receive the multicast. 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. This *shortest path tree* ensures the shortest possible delay in the delivery of any data packet, but it can create formidable routing requirements as the number of multicast groups and sources grow. The overhead at each router on a shared tree is $O(n \cdot s)$, where n is the number of multicast groups and s is the number of sources in the group

Other protocols assume a different approach in forming the tree and the means of initiating reception of 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 a single entry for each group. The router overhead is therefore $O(n)$, giving the shared tree approach superior scalability. However, because each such packet no longer travels over its shortest path to each receiver shared trees incur longer average delay in the delivery of a data packet.

PIM-SM and CBT differ with respect to the state in which they maintain their branches. Sparse mode PIM maintains branches in a *soft state*, in which not all senders are connected directly to the rendezvous point. Instead, the senders unicast packets to the rendezvous point, which then forwards the packets to all routers on the tree. A branch is not actually formed with a sender unless the traffic from that sender becomes very frequent. By relying

on unicast transmission for some packet traffic, PIM-SM avoids forming fixed branches that could become heavily burdened and instead relies on the underlying unicast routing algorithm to distribute packets over different links. In contrast, CBT operates in a *hard state*, such that all receivers and senders are connected to the tree by branches that stay in place once they are created. This allows orderly data packet delivery as long as the branches remain connected; loops in the underlying routing will not cause a packet to be lost or delayed.

During times of underlying unicast instability CBT can form loops. Loops in a shared multicast tree are fatal. When a data packet enters a loop, it circulates the loop endlessly. 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 starts forwarding so many packets to the rest of the tree that all links on the tree become saturated and are unable to function. The focus of this work is on improving on the Core Based Tree protocol to eliminate this looping problem and other problems that can keep a multicast tree from forming.

The next chapter describes the operation of the CBT protocol. Chapter 3 provides a complete description of OCBT. Chapter 5 discusses the performance of OCBT, which is analyzed by simulation and compared with CBT's performance. Chapter 4 shows the correctness and loop freedom of OCBT. Chapter 6 offers conclusions.

2. BACKGROUND

The Distance Vector Multicast Routing Protocol (DVMRP) [1] currently used in the Internet MBONE [6] forms a source-based tree from each sender and requires that each router maintain information for each source in each group. Additionally, it requires routers that do not wish to be part of the multicast group to explicitly send messages to prune themselves from the tree. Within a large network with a multicast group containing many sources, the overhead required by on-tree routers can be excessive.

CBT [3] [7] [8] [9] 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 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 a 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 message 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.

2.1 Looping in CBT

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, the tree will not always form in CBT. 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 2.1 shows the topography of a CBT sub-tree subject to this transient looping. The grey node is attempting to rejoin

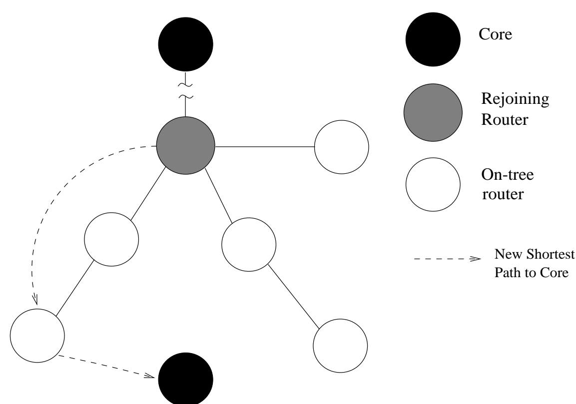


Figure 2.1: Looping in a disconnected CBT subtree.

along a newly formed shortest path to the next reachable core. Because its path passes through a descendent child, a loop will be 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 current CBT protocol specification [9], 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. This continual looping denies multicast service to the disconnected sub-tree, but it can be stopped if the rejoining router is allowed to flush the tree upon detecting a loop, after which each member of 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 by forwarding a flush message to all children of the router that is rejoining does not always solve the looping problem. In this case, flushing the tree can initiate a race condition in which local routers attempt to rejoin the core as it attempts to rejoin the primary core. Upon receiving a flush message, routers with members of their subnets

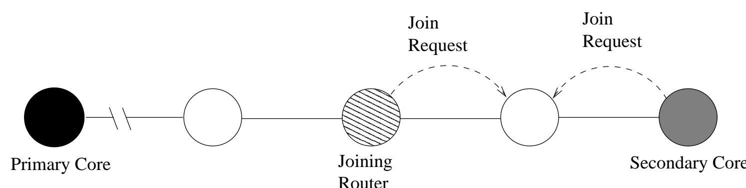


Figure 2.2: Deadlock or loop formation in tree formation

desiring the multicast immediately send a join-request on a hop-by-hop basis towards the closest core. That secondary core could be 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 2.2. 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 router 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 2.2, 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 occuring the forwarding of the rejoin-request back to the secondary router removes the loop, though the 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 is unstable during construction, the secondary core's attempt to join 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 acks 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 2.3. 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

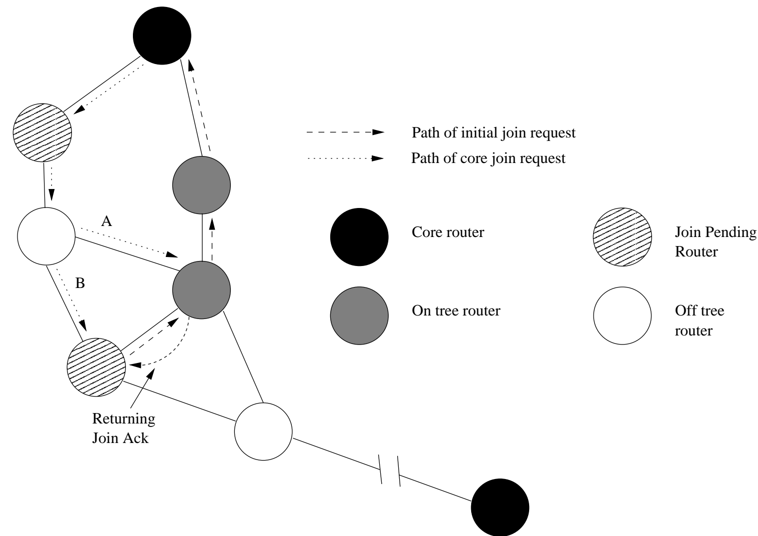


Figure 2.3: Undetected loop during tree construction under routing instability

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 2.4, 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

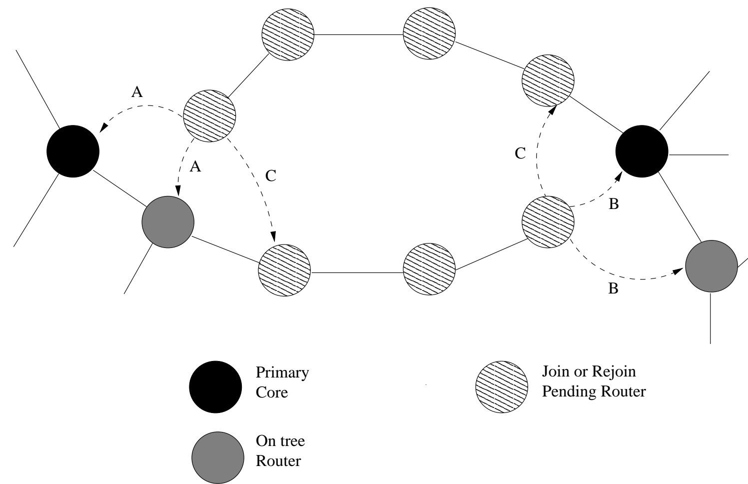


Figure 2.4: Permanent loops in the core backbone formed under routing instability

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 2.4 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. In contrast, OCBT maintains a hierarchy of cores with an arbitrary number of levels. Each core is assigned a *logical level*, which is a label indicating the cores place in the hierarchy of cores. By using this ordered scheme, OCBT completely eliminates the problems associated with CBT, reduces control traffic following a link failure, and allows for flexible core placement. OCBT does this without increasing the complexity of the protocol.

This chapter describes the differences between OCBT and CBT, discusses the difficulties of core placement, shows how an OCBT tree is built and maintained, and presents a formal description of the protocol.

3.1 Differences between OCBT and CBT

OCBT maintains a logical level for each node and core. 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. In contrast, CBT allows the formation of the loops as described in Section 2.1.

Join-requests in OCBT carry a field which contains the level a node must have to 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 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 [9] 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 the 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. To create a spanning tree takes $O(n)$ messages, 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.

3.2 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 in the United States has little use for information about local cores in Kazakhstan, 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.

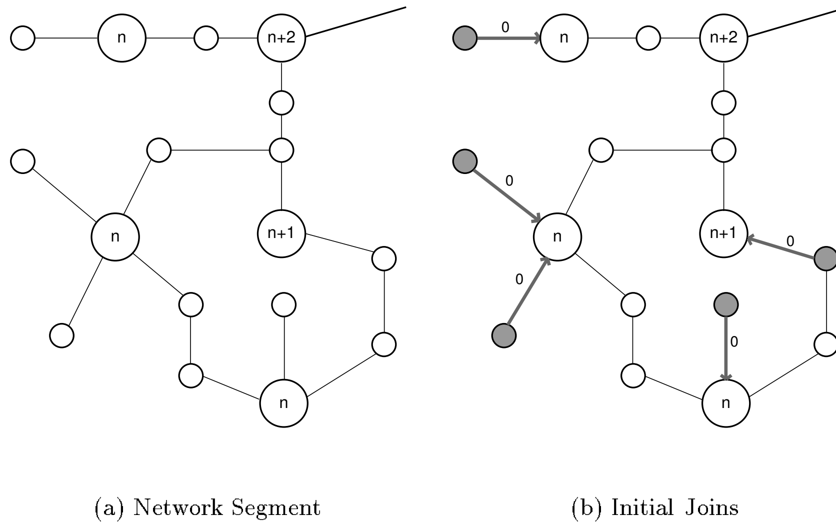
Alternatively, cores could follow a distributed scheme for disseminating their location and level, broadcasting or flooding their identity and location with an increasing time-to-live over each of their interfaces, or they could join a multicast group that existed solely for the purpose of core location 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], although this work remains unpublished apparently due to difficulties in defining the constitution of a scope and how to enforce scope limits. 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.

3.3 Tree Construction

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. These join-requests are stamped with level zero to indicate that the sending router can connect to any on-tree core or node safely. 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. Figure 3.1a shows a network segment which we will use to show the operation of OCBT. The larger nodes represent cores and are marked with the core level. Smaller nodes represent non-core routers, and are initially off-tree. Figure 3.1b,shows the gray colored nodes attempting to join their nearest core.



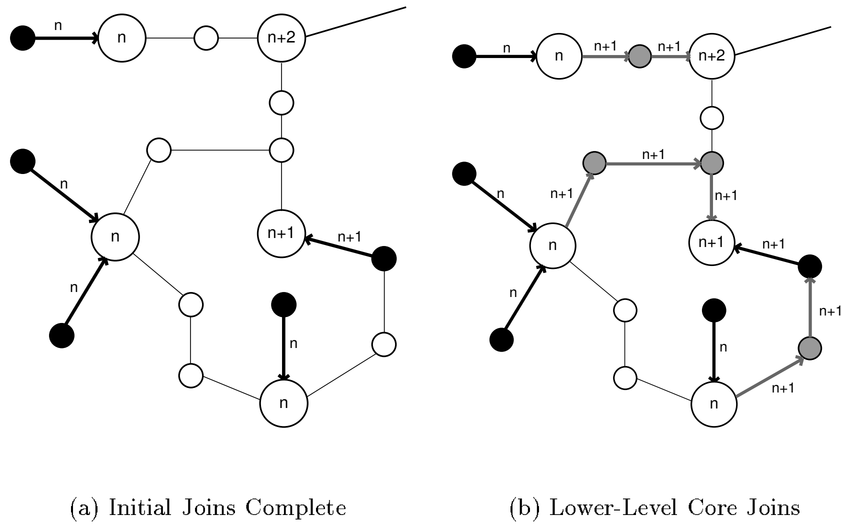
(a) Network Segment (b) Initial Joins

Figure 3.1: Initial OCBT Tree Building Process

Once the requests are received at a core, they are answered with a join-acknowledgment stamped with the core level. The acknowledgments travel the reverse route back to the sender of the request and bring each node up to the level of the join-acknowledgment. Figure 3.2a shows the state of the tree after these initial acknowledgments are received. For each initialized link, the arrow points from child to parent and indicates the level of the child.

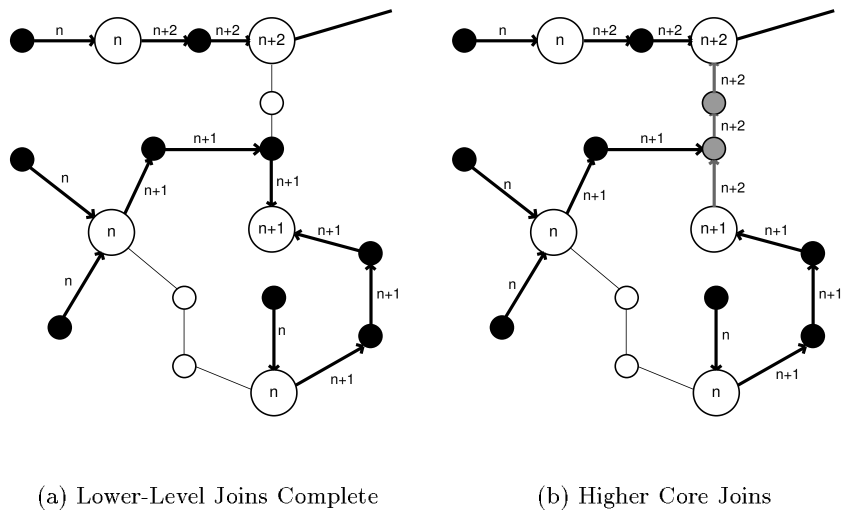
If a core receives a join-request and is not currently on the tree, it too must send a join-request towards the nearest core at the next higher level. The join-request is, in this case, stamped one level greater than the core level, indicating the lowest level of an on-tree node or core that can safely acknowledge the request and adopt the sender as a child. Figure 3.2b shows the process of the lower cores connecting to the higher.

Once a join-request reaches a node or core of appropriate level, the receiver again returns a join-acknowledgment stamped with the receiver's level. This reply again traces the reverse path and brings all nodes on the path up to the core level. This process is illustrated in Figure 3.3a. In this figure, the bottom-most requesting core reaches a child of the higher-level core, with the end result being an extension of the $(n+1)$-level branch back to the lower-level core. The top-most requesting core reaches a core of higher level than necessary,



(a) Initial Joins Complete (b) Lower-Level Core Joins

Figure 3.2: Building the tree from the Lower-Level Cores

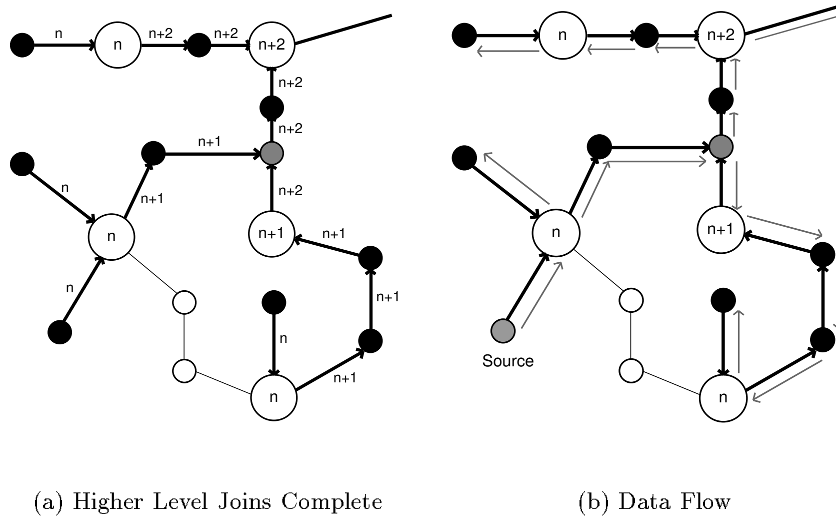


(a) Lower-Level Joins Complete (b) Higher Core Joins

Figure 3.3: Building the tree from the Higher-Level Cores

and a branch of that higher-level is built back to the sender. The center core reaches the higher-level core and a branch is built to it directly. Notice that in this branch, the level changes only at a core.

Once again, the $(n+1)$-level core will have to attempt to join the tree to provide service to its children. It sends a join-request towards the $(n+2)$-level core. As shown in



(a) Higher Level Joins Complete

(b) Data Flow

Figure 3.4: Build completion and Data Traffic Routing

Figure 3.3b, the first hop on the path to the higher-level core passes through a node that is already on-tree and which in this case happens to be a child of the sender. This does not create a problem, because the join request is for a node of level $n + 2$ and the child is only of level $n + 1$. The child quits from its parent and forwards the request, becoming parent to the $(n + 1)$-level core. The $(n + 1)$-level core, detecting that it will be connecting through a child, removes that child from its list of children. This prevents loops from forming over that one hop in the event the quit-notice is lost.

The new parent to the $(n + 1)$-level core does not flush its children by sending a notice down-tree to quit the tree, as it is expected that, once the join-acknowledgment is received, data packets can still be forwarded down the lower-level branch. This non-core node where the level changes is known as a *graft*, and occurs when a higher-level join request breaks a lower-level branch and the lower-level branch is maintained below the break. The level of a branch can only change at either a core or a graft. The graft node is illustrated in stripes in Figure 3.4a, which shows the tree after the join acknowledgment returns.

When the tree is constructed, data packets flow in the same manner as in core-based trees. A data packet from a source is forwarded to the parent, and from that parent over every other on-tree interface except the one leading back to the sender. In this manner the

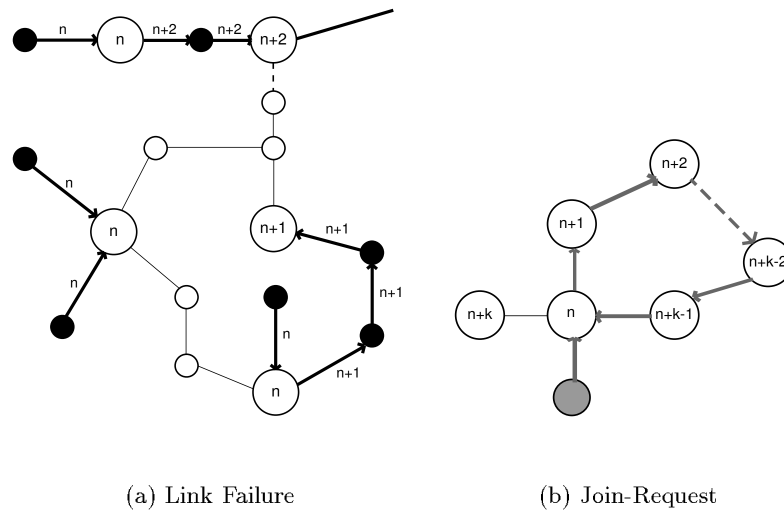


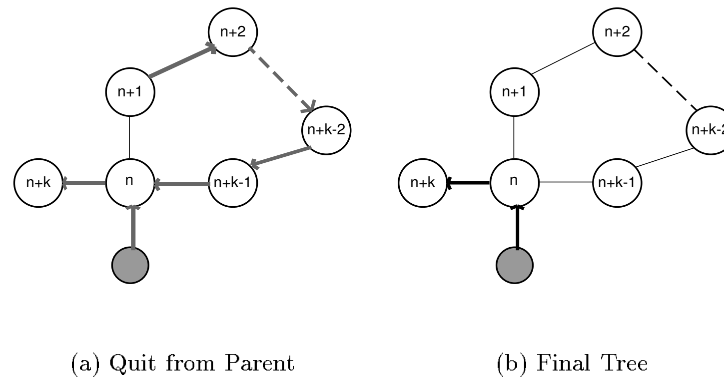
Figure 3.5: Link Failures

packet is forwarded to its parent and all other children as many times as required to reach the backbone. From there, it flows down every other branch to all on-tree nodes. This process is illustrated in Figure 3.4b. The gray node is taken to be the source and the flow of data packets is indicated by gray arrows.

3.4 Tree Maintenance

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 3.5a 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.

An interesting event can occur when lower-level branches are broken to make way for higher-level branches. A single router desiring the multicast can create a path spanning several higher-level cores without branching. If this path passes through itself at a lower



(a) Quit from Parent (b) Final Tree

Figure 3.6: Routing Loop Unrolling

level, the path can unroll and shorten. This can be seen in the series of figures below. First, in Figure 3.5b, the join-request follows some path and arrives at a node it has already passed through.

As the request is received at the n-levelnode, it is forced to quit from its parent and go to the higher level. It then forwards the join-request as in Figure 3.6a. As the $(n+1)$-level core now has no children, it is free to quit from the tree. Its quit-notice allows the $(n+2)$-level core to quit as well. This process continues until the $(n+k-1)$-level core quits from the n-level node. As the n-level node still has a child it cannot quit the tree and the final tree is much shorter than it would have been otherwise. Figure 3.6b shows the final tree. This process cannot occur completely if any of the $n+1$-level cores to $(n+k-1)$-level cores have any other children. The unrolling process halts at the first node with another child.

3.5 Network Partitions

In the event a link failure results in a network partition such that there exist router pairs that are out of contact, some special measures need to be taken to insure that a multicast tree is built within each partition and that the trees merge when the network reconnects. The network can form discontinuities in the logical level organization in four different ways: leaf nodes can be separated from contact with a core; a single n-level core can be unable to reach any of its sibling or higher-level cores; some singular n-level core or group of n-level

cores may not be able to reach any $n+1$-level core but can reach a $n+2$-level or higher-level core; or multiple n-level cores can be separated from contact with any higher-level cores.

In the first case, there is little to do. The leaf node simply must monitor its unicast distance to any reachable core to determine connectivity and wait until the network is reconnected to join the group.

In the second case, there is one highest-level core within the partition and the tree forms up to that n-level core. That core is not able to join any higher-level cores. It instead becomes the root of the tree formed within the partition until it is able to connect to a higher-level core. At that time, it joins the tree that formed within the other partition.

In the third case, one or more n-level cores attempt to join or rejoin the multicast tree and are not able to reach a $n+1$-level core. Upon failing to contact a $n+1$-level core, each lower-level core attempts to connect to any $n+2$-level that are reachable. If that fails, it increments the level sought and tries to connect again, until it connects to the higher-level core within the partition. In this case, the higher-level core will service the lower-level cores within the partition.

In the fourth and final case, a group of n-level cores are together in a partition serving the lover-level routers, but they are unable to reach any higher-level core. In this case, the same strategy used to build the highest-level backbone is used to connect the n-level siblings. An election mechanism is used to promote one member of the n-level group to $n+1$ level, at which point all n-level routers connect to the promoted core. The election mechanism is simple; the n-level core with the lowest IP address is chosen to be promoted. It is necessary for all n-level cores to agree on which cores are within the partition and which core is promoted; to this end the n-level core detecting the partition unicasts the list of cores it believes it can reach to every other reachable n-level core. When a n-level core receives one of these messages, it verifies that all of the cores on the list are unreachable, then returns its list of unreachable cores. As the unicast routing information propagates, then either all cores will have the same list of reachable cores, in which case the promotion occurs; otherwise the partition that was detected will turn out to be erroneous, in which

case off-tree cores reconnect to a higher-level core. The promotion information is limited to the n-level cores within the partition. The promoted core is responsible for monitoring the state of its unicast routing table to determine when the partition is resolved and the rest of the network is reachable. When this occurs, the promoted router flushes the temporary backbone, demotes itself, and rejoins the rest of the tree with its siblings.

3.6 OCBT Specification

This section formally specifies OCBT. OCBT's specification is divided into four sections. The first part is shown in Figure 3.7, and consists of the functions used repeatedly in the operation of the protocol. The operation of core nodes in response to control messages is shown in Figure 3.8, while that of non-core nodes is presented in Figure 3.9. The response of both core and non-core routers to a parent link failure or the expiration of a timeout timer is shown in Figure 3.10. Each part of OCBT's specifications follows the same conventions. 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 will reveal 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 the partition-recovery mechanism described above, either by returning the appropriate core or by starting the election process. 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.

```
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 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 /* this should only be reached with 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 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

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)
    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
```

Figure 3.7: Common OCBT Functions

```
On-Tree Core (message type, message level,source,
              core,originator)
  case (message type)
    Join-Request
      if (on child list (source))
      /* previous quit-notice must have been lost */
        Remove Child (source)
      if (source = parent)
        parent = null
        parent level = level
        process message as an Off-Tree Core
      if (message level <= level)
        Add Child (source, level)
      else
        if (message level > parent level)
          Break Branch (message level,
                        core, originator)
        else
          Add Child (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)
    Data
      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

Join-Pending Core (message type,message level,
                   source,core,originator)
  case (message type)
    Join-Request
      if (on child list (source))
      /* previous quit-notice must have been lost */
        Remove Child (source)
      if (source = parent)
        parent = null
        parent level = level
        process message as an Off-Tree Core
      if (message level <= level)
        Add Child (level)
      else
        if (message level > parent level)
          Break Branch (message level,
              core,originator)
        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)

    Join Ack
      if (source = parent) and (message level > level)
        parent level = message level
        Multicast Message (Join Ack, message level)
        state = On-Tree Core

    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 */
        Join Tree (level)
```

Figure 3.8: OCBT Protocol for Core Nodes

```
On-Tree Router (message type, message level,
                source, core, originator)
  case (message type)
    Join-Request
      if (on child list (source))
        Remove Child (source)
      if (source = parent)
        if (message level > level)
          Break Branch (message level,
                        core, originator)
        else
        /* a flush message was lost */
          Forward Message (Flush Message, parent)
          for each child on list
            Remove Child from List (child)
          parent = Next Hop (core)
          level = message level
          Send Message (Join-Request, parent,
                        level, core)
      else
        if (message level <= level)
          Add Child (level)
        else
          Break Branch(message level,
                       core, originator)
    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)
    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 Router (message type, message level,
                     source, core, originator)
  case (message type)
    Join-Request
      if (on child list (source))
        Remove Child (source)
      if (source = parent)
        if (message level > level)
          Break Branch (message level,
                        core, originator)
        else
        /* a flush message was lost */
          Forward Message (Flush Message, parent)
          for each child on list
            Remove Child from List (child)
          parent = Next Hop (core)
          level = message level
          Send Message (Join-Request, parent,
                        level, core)
      else
        if (message level <= level)
          Add Child to List (source)
        else
          Break Branch(message level,
                       core, originator)
    Join Ack
      if (source = parent) and (message level >= level)
        level =message level
        Forward Message (Join Ack, level,source)
    Quit-Notice
      if on child list (source)
        Remove Child (source)

    Flush Message
      if (source = parent)
        Forward Message (Flush Message, level, source)
        Remove Children (0)
        /* only reached if above function returns */
        level = 0
        Join Tree (level)
```

Figure 3.9: OCBT Protocol for Router Nodes

```
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
      else
        parent = Next Hop (Find Core(level + 1))
        Send Message (Join-Request, parent, level, 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)
        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

    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 3.10: OCBT Protocol for Timeouts and Parent Link Failures

4. CORRECTNESS OF OCBT

In this chapter we show that the OCBT protocol is correct, which implies that the OCBT protocol provides a correct multicast tree within a finite time after a router starts creating the tree or an input event forces the tree to be modified, and that the OCBT protocol never creates multicast loops.

It is obvious that no protocol can create a multicast tree across disconnected components of a network. However, temporary network partitions may occur. We first limit our proof of correctness to the case in which the network is connected, i.e., we assume a connected network in which there exists some physical path between any pair of nodes that can become part of the multicast tree. After presenting this proof, we extend it to deal with network partitions.

4.1 Connectivity in a Connected Network

First, we demonstrate that the OCBT protocol never deadlocks and terminates in a finite time producing or modifying a tree.

Showing that OCBT has no deadlocks is straightforward, because all messages that require a reply in the protocol are sent hop-by-hop or unicast. Accordingly, deadlocks are prevented in the OCBT protocol by means of a time-out counter initiated whenever a message is sent for which a reply is expected. In the event that the time-out counter decrements to zero, the message is re-sent if the node is the initiator of the request; otherwise, the node goes back to the off-tree state, rejoining as necessary to provide service to subnet members or established children. After a maximum number of retransmission attempts, the sender can simply assume that the intended receiver cannot be reached.

The following theorem demonstrates that the OCBT protocol terminates with the correct result when the network is connected.

Theorem 1: Given a connected network, the OCBT protocol terminates within a finite time with the correct result.

Proof: The proof proceeds by contradiction and assumes that no deadlocks occur in any message exchange. The assumption is valid as any deadlock would eventually result in a retransmission of the original message upon timer expiration.

Any individual node must join a core in order to join the tree. The node's join-request travels towards the closest core, and the join acknowledgment, returning from the core or some member of its tree, creates a branch to the node with a logical level equal to the core's level. If the core is not connected to the multicast tree, then it connects to the next higher-level core. Each core connects to the next higher-level core, until the backbone is reached. For the OCBT protocol to fail in establishing a multicast tree, the join-request must fail.

Assume that a core attempts to join to the next higher-level core and fails at some point in between. Three things can happen to a failed join request. First, the request reaches the higher-level core or a branch of the tree at the higher-level core's level. In this case, a join acknowledgment is sent back to the lower-level core, extending the branch to the lower-level core and violating the initial assumption. Second, the request is received at an on-tree node of level lower than the request. In this case the receiver quits from its parent and joins the branch that is being built, the branch below the receiver is maintained and because the branch below is not flushed it does not effect overall construction of the tree. The request is forwarded on the next hop to the intended core. Finally, the request reaches an off-tree node, which is forced to forward the join request. The request must then continue through some path of off-tree and lower-level nodes until it reaches a higher-level node or core and is acknowledged. It is clear that in no case is the branch prevented from forming, violating our initial assumption; therefore the theorem is true.

4.2 Loop Freedom

We now show that the OCBT protocol maintains a tree structure at all times, that is, that a data packet introduced onto the tree is never received by any node more than once, even if the underlying unicast algorithm forms transient routing loops before converging.

We divide the proof into several parts. In Lemma 1, we show that if a connection is completely formed between a child and parent, that is, the parent and child have consistent state information about the connection, the logical level of the child is always less than or equal to that of the parent. Lemma 1 also shows that if the state information is incorrect, that is, if one of the child-parent pair believes the connection to be in place but the other does not, then the state information will be corrected. Lemma 2 shows that when the state information is consistent, no loops can form among nodes of varying level. Lemma 3 derives the same result for nodes of the same level. Lemma 4 demonstrates that even if the state information between a parent and child is inconsistent, no loops will form in the data transmission path. This final result is important to show that loops will not form during the time that it takes OCBT to correct the state information.

The proof presented in Lemmas 1 through 3 and Theorem 1 below is based on the fact that OCBT always dissolves a lower-level branch of a tree before allowing a higher-level branch to be built in its place. This process is termed coercion, as a higher-level message coerces a lower-level node to higher level. By contradiction to the fact that a child in OCBT always has a logical level that is smaller than or equal to its parent's level, the proof first shows that with consistent state information, loops cannot form between nodes of different levels. The proof then shows that loops can only form when the underlying unicast algorithm contains loops in its routing table, and that under these conditions the OCBT protocol does not introduce loops into the tree. The following lemmas depend on the fact that lost messages will eventually cause a timeout and cause a retransmission; lost packets are therefore ignored.

Lemma 1: In an OCBT, a child node always has a logical level smaller than or equal to its parent if the state information between child and parent is consistent. If the state information is not consistent it will eventually be made consistent through the use of the keep-alive mechanism.

Proof:

To prove the Lemma, we first examine parent-child connections between non-core nodes.

For each of the four connection setup and two modification possibilities, we show by contradiction that the parent will be of level equal to or greater than its child. Assume for each case that a connection between two non-core nodes exists and that the child is of higher level than the parent.

Case pc-1: The connection is formed by two nodes initially off-tree (with no level). In this case, the child has a subnet member wishing to receive the multicast and sends a join-request of level zero to its parent. Its parent forwards the join-request towards some core, which receives it and returns a join-acknowledgment with the core level, which is greater than zero. The parent then changes level to the core level and forwards the ack to its child, which also changes level to the core level. They both then have the same level, which violates the initial assumption of the child being of higher level. As the child ignores any join-acknowledgments from other than its parent, such a connection could not form this way.

Case pc-2: The connection is formed between a child off-tree node (with no level) and a parent on-tree (with level greater than zero) node. The child sends a level zero join-request to the parent. The parent receives it and sends a join-acknowledgment with its own level in it. The child then sets its level equal to the parents level, again violating the initial assumption.

Case pc-3: The connection is formed between a join-pending child (with some level) and an off-tree parent. The only way the child could have level was if it is forwarding a join-request from some core down tree, in which case the message is one logical level higher than the lower-level core. When the forwarded join-request reaches the parent, the parent sets its level equal to that of the child and again forwards the message. When the ack is received it is of level equal to or greater than the level of the request. If the level of the acknowledgment is higher then the parent level, then the parent raises its level to that of the ack and sends it to its children; otherwise it maintains the same level and forwards the acknowledgment. If the child receives an ack of higher-level than its level, it also raises its level, otherwise it maintains the same logical level. As the parent and child either both

raise to the level of the ack or remain the level of the request, they both have the same level and violate the initial assumption.

Case pc-4: The connection is formed between two nodes with some initial level. As before, the child could only have level by forwarding a request from a down-tree core. The parent could either be on-tree or in a join-pending state. If the request from the child is of lower or equal level than the parent, the parent sends an acknowledgment with its own level and the child's level is changed to the parent's level. In this case both the child's and parent's level are equal and the initial assumption is violated. If the request from the child is of higher level than the parent, coercion occurs and the parent quits from its parent, switches to the child's level and forwards the request. The rest of the case is as described above — the returning ack dictates the level of both child and parent, and the initial assumption is violated.

Case pc-5: Initially, the parent is of level equal to or greater than that of its child, but the parent's level is decreased. There are two ways the parent could reduce its level. First, it could receive quit-notices from all its children and go off-tree, but as it still has one child this is clearly not possible. Second, it could receive a flush message from its parent. In this case it forwards the flush message to all its children, clears its child list and changes its level to level zero. If the child receives the flush message, it also decreases its level to zero and leaves the tree, so therefore the flush message from parent to child must have been lost and the child remains at a higher level than the parent. Now the parent either stays at level zero or rejoins the tree at a lower level than its child and the state information is inconsistent between child and parent. The keep-alive mechanism acts here to force the child to a lower level. After the child's keep-alive packet is ignored by the parent who no longer considers the child part of its child list, the child follows its link failure procedure and lowers its level to zero and attempts to reconnect. We then would have one of cases 1 through 4 above when the child attempts to reconnect to its parent, and the state information would once again be correct. If the parent attempts to connect to the tree again through its former child before the keep-alive mechanism acts to lower the child level, the join-request from

the parent serves as a implicit retransmission of the lost flush message; the child forwards a flush message to its children and joins the forming branch as if it were off-tree.

Case pc-6: Initially, the parent is of level equal to or greater than that of its child, but the child's level is increased. The child's level can only increase due to receipt of a join-request from a down-tree core with higher level. When the child receives the request it immediately quits from its parent, removes its parent information and changes to the higher level. If the parent receives the quit-notice then it removes the child from its child list and sends a quit acknowledgment breaking the link and making the state information consistent. Hence, there are four possible sub-cases - the quit-notice is either received or not, and the next hop in forwarding the join-request from the child is either through its parent or not. If the quit-notice is received and the next hop is not through the parent, then the initial assumption is violated as there is no link between the child and parent. If the quit-notice was received and the next hop is through the parent than either case 3 or case 4 applies. If the quit-notice is not received and the next hop is through the parent, then when the parent receives the request it removes the child from its child list then it too changes its level to that of the request and ends with the same level as its child, violating the initial assumption. Finally, if the quit-notice is not received and the next hop is not through the parent, then the keep alive mechanism will cause the parent to remove the child from its list, breaking the link, making the state information consistent and ruining the initial assumption.

In considering cases involving cores, it is simpler to replace a core with a simple model that reflects its operation. Each core can be viewed as two nodes together - a child node of fixed level equal to the core level and a parent of variable level at least one greater than the core level. We will refer to these model nodes as the *bottom core* and the *top core*. This reflects the way that a core has a fixed level (the bottom core), while still being able to attach itself to higher-level branches (the top core). Messages coming to the core can be viewed as going to one or the other of this pair depending on the message level.

We will again use contradiction to show that a child cannot be of higher level than

a parent in connections between cores and non-cores. First assume that there exists a connection between a core parent and a non-core child and that the child has level higher than the top core.

Case c-1: The child has no level and connects to the core. It does this by sending a level zero join-request to the core. When the core receives the join-request it is handled by the bottom core as the the message level is less than the core level. If the core is in an on-tree state, it simply replies with a join-acknowledgment with the core level. When this reaches the child, the child changes to the logical level of the bottom core, which violates our assumption. If the core is in the off-tree state it sends a core-level join ack which raises the level of the child to the level of the lower core. The top core is then forced to connect to a higher-level branch. If the join request from the top core passes through the child node, then the child is forced to quit from its parent and go to the level of the top core pending receipt of a join-acknowledgment. In quitting from its parent it becomes parent to the core and hence violates our initial assumption. Even if the quit-notice is lost, the bottom core to removes the child from its child list when the join-request is sent, again breaking the link, restoring the state information and violating the initial assumption.

Case c-2: The child has some level and connects to the core. If the child level is less than the core level, then the bottom core sends a core-level join-acknowledgment which raises the child to the same level as the parent and violates the initial assumption. If the child level was higher than the core level then the top core receives the request. If the child level is greater than the level of the top core then the top core is coerced to the child level, violating the assumption. If the child level is less than or equal to the top core level but greater than the lower core level, it receives a join ack equal to the top core level, which coerces the child to the same level as the top core, again violating the initial assumption.

Case c-3: The lower level child is connected to the core but changes level. The only way the level could be raised is if the child receives a join-request of higher level and is coerced to that greater level. When it receives that higher-level request, it immediately sends a quit-notice to its parent. If that is received then the link is broken, which violates

the initial assumption. If the quit-notice is lost, then the keep-alive mechanism eventually realizes that the child is not responding to the parent and causes the link to be broken by the parent, restoring the incorrect state information. If the next step in the join-request is back through the parent, then case 2 above applies.

Case c-4: The child is of lower level than the core but the core decreases its level. The only way the core can decrease its level is by receiving a flush message from its parent. In this case the upper core decreases its level to one greater than that of the bottom core. If the child is of level equal to or less than the bottom core, then it is not forwarded the flush message but remains attached, and its level rules out our initial assumption. If the child is of level greater than the bottom core but less then or equal to the level of the upper core, it is forwarded the flush message which drops its level to zero, violating the initial assumption. If the flush message is lost, the keep-alive mechanism from the child to parent detects the state change and forces the child to drop the link to its presumed parent and become level zero, flushing its children. Dropping the link to the parent again violates the initial assumption and repairs the state information.

We now examine core children connected to node parents. Assume that there exists a connection between a core child and a non-core parent and that the top core which connects to the parent has level greater than that of the parent.

Case cn-1: The connection is formed between an off-tree core child and an off-tree node parent. The core sends a join-request of logical level at least one level greater than its level, coercing the off-tree parent to that level and placing it in a join-pending status. When the join-acknowledgment is received it is also at least one level higher than the core level. The returning ack dictates the level of the parent off-tree node and child top core node; they will be the same so the assumption is violated.

Case cn-2: An off-tree core is connected to an on-tree node. If the node is of level lower than the top core, then it is coerced to the same level by the join-request. The join-acknowledgment then makes both the node and the top core the same level when received, violating the initial assumption. If the node is of level equal to or greater than the top core,

it immediately sends an ack with its level, raising the top core to the level of the on-tree node, violating the initial assumption.

Case cn-3: Initially the core is of level equal to or less than the node parent, but the core raises its level. The core could only raise the level of the top core, and only in response to a join request which carried a higher level than the top core. When the core receives this request it is forced to quit from its parent. If the next hop is not through its parent and the quit-notice is lost then the keep-alive mechanism forces the breaking of the link and correction of the state information. If new next hop is through the former parent, then the parent detects that the quit message was lost and corrects the state information before processing the request. If the request is then of level greater than that of the parent, it coerces the parent to level of the higher node and the join ack places both child and parent at the same level, violating the initial assumption. If the join-request is of lower level than the parent, then the parent returns a join ack of its level and the top core again becomes the level of the parent.

Case cn-4: The parent node initially is of level equal to or higher than its core child, but lowers its level. The only way the parent lowers its level is in response to a flush message, at which time it would forwards the message to all its children and goes to level zero. If the flush message is received at the core then it severs its link to the parent and the initial condition would not be possible. If the flush message does not reach the core then the keep-alive mechanism eventually causes the state information to be restored. Once the link is broken, the core would attempt to reconnect to the tree. If the core tries to go through the same parent then one of cases cn-1 or cn-2 would apply. If the former parent attempts to reconnect to the tree though the core before the state information is corrected by the keep-alive mechanism, then the core detects the loss of the flush implicit in the join-request, corrects the state information and handles the join-request and in case c-1 or c-2 above.

The last type of connection we need to consider is direct core to core connections. Once again assume that we have two cores, one a child of the other, and that the child has a higher level.

Case cc-1: Both cores start in an off-tree state. The child receives a join-request that is either lower or equal to core level, or is higher than core level. If it is lower than core level the child forwards a join-request one level higher than the bottom core, otherwise the child forwards a join-request more than a level higher than the bottom core. When this request reaches the parent, if it is lower in level or equal in level to the parent's bottom core, it is received there and immediately acked with the bottom core level. The top core level of the child will then be raised to the bottom core level of the parent. As an upper core always has a logical level higher than the bottom core, this means that the highest level of the child is equal to the lowest level of the parent and the initial assumption is incorrect. If the join-request is higher in level then the parent's bottom core, it is handled by the parent's top core, which is coerced to the level of the request. When the parent forwards the join-request and receives an ack, it forwards that ack and the top core of both parent and child will have the same level. As the level of the top core is always higher than that of the bottom core, the assumption is again violated, as the connection that is formed between the top cores of both parent and child has equal level.

Case cc-2: The child starts in an on-tree state and the parent in an off-tree state. The only case in which an on-tree core attempts to join an off-tree core is if it is reconnecting after a flush or if the keep-alive mechanism has detected a failure. Both of these cases are functionally equivalent to case cc-1, except that the join request is definitely one level greater than the child core.

Case cc-3: The connection forms between an off-tree child and an on-tree parent. The child sends a join-request with level greater than the its bottom core level. When this message arrives at the parent node, it is either acked by the bottom core of the parent, in which case the top core of the child becomes the same level as the bottom core parent, violating the initial assumption, or it is handled by the top core. If the request is of higher level than the top core, the parent's top core undergoes coercion, and upon the receipt and forwarding the ack, the top core of both child and parent are equal, violating the assumption. If the request is of level lower than the top core of the parent, it is acked with

the parent's top core level and both top cores are of equal level, again showing the assumed connection could not occur.

Case cc-4: An on-tree child connects to an on-tree parent. The only case in which an on-tree core connects to another on-tree core is if it detected a link failure or received a flush message. In this case the child goes off-tree and attempts to reconnect, leading to the situation of case 3.

The lemma then follows, because in every possible case of a connection being formed, none can result in a child having greater level than its parent.

Lemma 2: No loops can form in an OCBT with nodes of varying level when state information is consistent.

Proof: If a loop was to form with nodes of differing level, then at some point a connection would be made from a node of lower level to a node of greater level. At another point in the loop a connection with a corresponding decrease in level would have to form. From Lemma 1, a child cannot have level greater than its parent, so such a loop is impossible.

In fact, if one was to attempt to form such a loop, coercion would make the loop all one level. If a loop is to form, it must occur with nodes all of the same level. We will show two results here. First, that loops of just one level can only from when the underlying unicast algorithm from loops that are established in its routing table, and second, that the OCBT protocol will prevent them from becoming established in the structure of the tree.

Lemma 3: No loops can form with nodes all of the same level when state information is consistent.

Proof: Assume that the underlying unicast algorithm is stable and does not contain loops, and that a loop has formed in the OCBT Tree. This loop must be between nodes of all the same level, as shown in lemma 2. The loop must be formed when some core sends a join request towards a specific higher level core. The loop forms when the request passes through some series of non-core nodes (which may have been coerced to the same level as the request) and top core nodes (the level transition from bottom core to top core would have instanced a level change, which by lemma 2 is not possible) and returned to the original sender or one

of the nodes in the path the request had already traced. That the shortest path from one specific node to another specific node passed through itself violates the assumption that the underlying unicast algorithm was stable and loop free. Therefore such a loop could not occur.

Now assume that the underlying unicast algorithm is unstable and contains loops in the routing table while converging. We will use proof by contradiction to show that a loop cannot form with nodes of all the same level, even in these circumstances. To reduce the number of cases we need to examine we make the observation that as all nodes have to be of equal level, we can treat cores as simply the upper core, as any connection to the bottom core would require a raise in level to depart through top core. Assume that a loop exists with all on-tree nodes:

Case 1: It is formed of all non-core nodes. There are two ways this could form. First, some node with a subnet member wishing to receive the multicast sends a level zero request that goes through a number of other non-core nodes and, because of new instability, arrives back at one of the initialized nodes in the join-pending state. At this point, the node the request returned to accepts the request as being of the same level and awaits its join ack before replying to the request. The network will stall here, as no join-acknowledgments are ever sent. Eventually, the original sender will time out and attempt to connect again. If by this time the unicast loop is not resolved then the node wishing to connect would not receive a return to its join-request and would try to connect to another core. No loop exists so our initial assumption was incorrect in this case.

If the loop formed with all non-core nodes carrying a request from a core and having a non-zero level, a similar process occurs. As the message returns to a node where it had already been received, the request stalls pending a join ack. As the nodes in the loop time out they do not attempt to rejoin but instead go back to their off-tree state.

If multiple nodes in the same loop attempt to join at the same time, each receives the others' join-request and holds them until they time out. Again, no loop would form. As loops will not occur with all non-core nodes, any possible loop would include a core in the

loop.

Case 2: The loop consists of core and possibly non-core nodes. In this case the loop is started in one of two ways. First, any core sends a join-request. As the request traverses the loop, each non-core node is coerced to the request level. Later, the request arrives back at the sending core. At this point the core detects the loop by reading the original sender ID carried in the request, realizes it was the original sender and takes steps to cancel the loop by quitting from its parent and sending a flush message to all its child. Again, the core attempting to join the tree will fail to connect to the higher-level core until the unicast loop is removed.

Second, the join-request comes from a core outside the loop through a non-core node. As it circulates the loop it coerces non-core nodes and top cores to higher level. When it closes the loop, the non-core node again stalls the protocol until all nodes time out and the original sending core attempts to join again, possibly to another core.

If two or more simultaneous join-requests are circulating in the loop one of two things occurs: if all requests are of the same level, then the protocol stalls until time outs occur, no matter where the requests originate. If the requests are of differing levels then the highest level request coerces all nodes in the loop to that highest level, terminating in a time out at a non-core node or in detection at a core. In either case no loop forms, and the initial assumption is contradicted.

Lemma 5: The OCBT protocol does not introduce loops into the multicast tree when the state information about a link between a child and parent is inconsistent between the child and parent.

Proof: We have already shown in Lemma 2 that, due to the loss of a quit-notice or a flush message, the information about the state of a link can differ between a parent and child. We have also shown that in every case in which this occurs, the keep-alive mechanism or receipt of a new join-request from a parent or child causes the information to be made correct. We will now show that loops cannot form during the transient period of incorrect state information. The proof is again by contradiction of the assumption that a loop exists

as a result of incorrect state information.

Case 1: The loop is formed when a parent node maintains information about a child after that child has sent a quit-notice that was lost, and that child became a descendent of the parent upon rejoining the tree. In this case any data packet reaching the parent, child or any intermediate nodes formed on the branch between them will be forwarded to all nodes on the branch. In addition, a copy will be sent over the link from parent to child. If this packet were accepted by the child it would then be forwarded up the branch to the parent, completing the loop. However, since the child does not consider the parent its parent, the data packet is dropped and no looping occurs, violating the initial assumption. As the child no longer sends keep-alive packets to the parent, the incorrect state information about the child maintained by the parent will be corrected once the parent detects the absence of the keep-alive messages.

Case 2: The loop is formed when a child node maintains information about a parent after that parent has sent a flush message that was lost, and that parent became a descendent of the child upon rejoining the tree. Again, any data packet reaching the parent, child or any intermediate nodes formed on the branch between them will be forwarded to all nodes on the branch. The child would then send a copy to the parent. Again, as the parent no longer maintains information about that child, the data packet is dropped and the data packet is not sent around the loop. The incorrect state information maintained by the child is corrected when it does not receive replies to its keep-alive messages.

Case 3: The loop is formed when a parent node maintains information about a child after that child has sent a quit-notice that was lost, and that child reconnects directly to the parent. In this case it is actually impossible that the loop forms, as every on-tree or join-pending core or node checks the source of a join-request against their child list; if the sender of the join-request is on the child list, it can be assumed that the quit-notice from child to parent was lost and the incorrect state information is removed before processing the join request.

Case 4: The loop is formed when a child node maintains information about a parent

after the parent sent a flush message to the child and that message was lost, and the parent connects directly to the child. Again, this type of connection could not occur as every on-tree or join-pending node checks to see if its presumed parent is the source of the message. If the parent is the source and the message level is lower than the level of the child, a flush message must have been lost. The child then removes the incorrect parent information, forwards a flush message to its children as appropriate and joins the branch that is forming.

As no loops form in any of the possible cases of inconsistent state information, no transient loops form.

Theorem 2: The OCBT protocol never introduces loops into a multicast tree.

Proof: The proof is immediate from Lemmas 3,4 and 5. If loops of varying levels are not formed,loops of the same level are not formed, and loops are not formed when state information is inconsistent, no loops can be formed.

4.3 Connectivity in a Partitioned Network

Given that OCBT forms a tree and does not loop in a connected network, the question next arises as to how the protocol performs in a partitioned network. We show here that, even if the network is partitioned, each network component that contains at least one core forms a tree without looping, and that within a finite time after the network is reconnected, the multicast tree reconnects.

Theorem 3: In a partitioned network, OCBT forms a multicast tree within each partition and reconnects each tree within a finite time after the partition is removed.

Proof: It is first important to note that, if there is a partition in the network, then each of the connected components forms a network in and of itself. If the logical ordering is maintained so that all children are of level less than or equal to their parent, then there will be no violation of any of the lemmas in Section 4.1 and 4.2. To see that this does not occur, consider the possible ways in which a network partition could separate the logical levels we have imposed.

Following a network partition, the logical levels in OCBT can be separated in one of four ways in each half of the component. For each case we consider the highest-level core available in the component.

First, a leaf node can be separated from all of the cores. Second, a n-level core can be separated from its siblings and any higher-level core. Third, one or some n-level cores can be separated from any $(n+1)$-level to $(n+k)$-level cores, but be able to reach a $(n+k+1)$-level core. Fourth, a group of n-level cores can be unable to reach any higher-level cores. In each case we consider only one half of the partition as the other half must fall into one of these four cases as well.

Case 1: In the first case we are presented a network component containing no cores and another containing all cores. The smaller network component will not be able to build a tree, as it contains no cores. Since no join-request that is sent can be acked due to the absence of cores to ack them in the network component, no branches and hence no loops can be formed. The individual leaf nodes will continue to attempt to reach a core until such time as the partition is removed. When the network is whole we have shown in Section 4.1that leaf nodes can join the multicast tree within a finite time in a connected network with cores.

Case 2: In the second case, we are given a component containing a single core of level n and at least one lower-level core of each level down to some lower level or to the leaf nodes. In this case, the network is still properly ordered; a tree can form up to the n-level core. This n-level core will then have the responsibility of monitoring its unicast routing table to determine when the partition is repaired and of joining to a higher-level core or a sibling on the other side of the partition. Because the ordering of the logical levels are not violated, the tree will be built and will reconnect within a finite time after the network is reconstituted.

Case 3: In the third case, there are cores of levels up to level n and from level $(n+k+1)$ in the component; however, all cores of levels between $(n+1)$ and $(n+k)$ are on the other side of the partition. In this case, even though the core ordering in not continuous, the logical order is preserved. The n-level cores are able to connect to the $(n+k+1)$-level

cores once they realize they are unable to reach any core between those levels. This extends branches of $(n+k+1)$-level down to the n-level cores. Once the network reconstitutes and the components are reconnected, it follows from sections 4.1 and 4.2 that the cores at the highest level connect to cores in the other partition within finite time to form one single tree.

Case 4: In the final case, the OCBT protocol takes specific action to preserve the logical ordering. In this case, the component contains a group of n-level cores with no higher-level cores present and it is not immediately clear how the tree should form as there is no single highest-level core present. Therefore, after the n-level cores have failed to reach a higher-level core, they undergo an election process to promote one of their members to level $n+1$. This election process results in all the n-level cores selecting the same core to be promoted within a finite amount of time. As each n-level core exchanges a list of reachable cores with all other n-level cores within the component, when the underlying routing has converged within the partition, all n-level cores within the component are aware of each other. This happens within finite time as the underlying routing must converge in finite time. Once all n-level cores have the same list of n-level cores, all follow the same election mechanism in promoting the core with the lowest IP number, so a single n-level is chosen to be promoted. Once this core has been promoted to level $n+1$, the other cores join with it forming a tree. The logical ordering is now correct.

When the network is reconstituted, the promoted core detects that it is able to reach a core at level $n+1$ or higher from the other part of the partition. At that point it flushes the tree down to the n-level cores, demotes itself back to level n and attempts to rejoin the full multicast tree with it siblings. Because the logical ordering is preserved throughout this process, it follows from the sections above that a multicast tree forms within the partition without forming loops and that a full multicast tree if formed within finite time after the network is reconstituted.

5. PERFORMANCE OF OCBT

To examine the performance of CBT and OCBT in a realistic manner,we created a simulation of each protocol using a simulation package[1] that supports protocol layering. These simulations ran on top of a Bellman-Ford unicast routing layer and used routing information from the unicast layer. With this simulation we measured the end-to-end delay of data packets traversing the tree, measured the number of messages of each type sent before and after a link failure, and recorded the formation of transient loops in CBT that required explicit action to remove. In addition, each case in which a CBT sub-tree was unable to reconnect to the tree, as shown in Figure 2.1, was recorded. We found no mechanism to determine the formation of loops that CBT would not detect; if it were simple to do this we could recommend a way to detect and prevent these loops in the CBT protocol. 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 5.1, 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 occured four times as frequently as the CBT and OCBT keep-alive messages. 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

[1] The protocols presented in this paper were implemented, tested, and analyzed with the help of the C++ Protocol Toolkit (CPT) by Rooftop Communications Corp. of Los Altos, CA. Contact Rooftop at info@rooftop.com or 415/948-2720 for further information on the CPT.

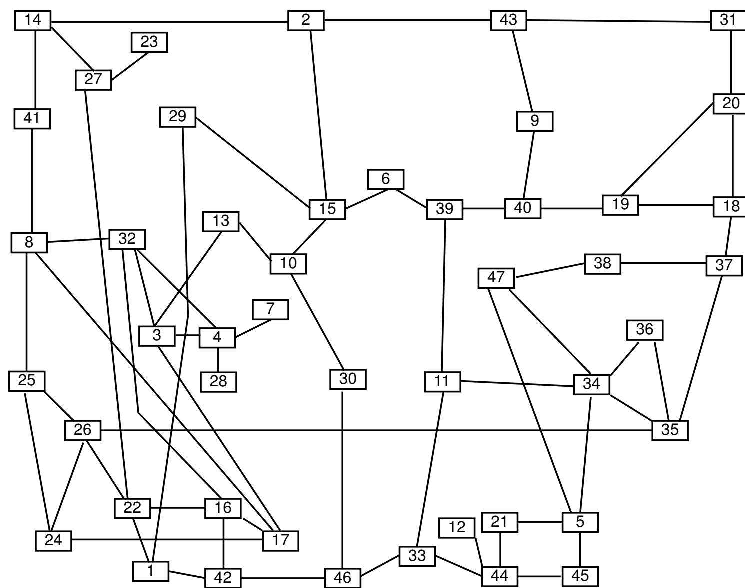


Figure 5.1: Arpanet Simulation Topography

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.

We tested four different implementations of CBT. The current specifications for CBT do not call for the sub-tree to be flushed when a loop is detected. In cases where a loop is detected at a non-core node, however, flushing the sub-tree can prevent the re-creation of the loop. We therefore included the flush message on loop detection in two of our implementations. Older versions of the protocol also included a *core-ping* message that was sent to ensure that a path existed to the core to which a router was attempting to connect. This was removed to promote shorter join latencies. This message could serve to better CBT behavior in the case of a unstable network, however, as no branch would be built until a relatively stable path existed to the core. We therefore included the core-ping in two of our simulation sets to see if putting it back in the protocol would improve its performance

under failure. In the four different implementations studied, one had no core ping or flush on loop detection; one had no core-pings but did have flush on loop detection; the other two implementations were the same as the first two but included the core-ping.

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.

5.1 OCBT and CBT

First we compared the performance of OCBT against that of the current CBT protocol [9] over the 12 different core sets shown in Figure 5.1. The results are summarized in Table 5.2. Each run shows the averaged 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 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 2.1. 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

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 5.1: Cores used in simulation

runs, we found a disconnection rate of 11.3% under the current protocol specifications [9]. 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

Run Number	Build Messages	Repair Messages	Average Delay	Delay Variance	Transient Loops	Disconnected 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 Based	72		1.00	1.00			
95% Confidence Interval							
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 5.2: OCBT vs. CBT

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 will eventually 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.

5.2 CBT implementations

We next investigated the different implementations of CBT to see if using core-pings to verify a path to the core or if flushing the sub-tree upon loop detection would improve the looping performance of CBT. The results of the non-standard implementations are shown in Figure 5.3. Implementations using a core-ping are so designated by the addition of a CP suffix, and those that flush the tree use an F suffix.

The results of our simulations show that none of the simple changes made in the implementations of CBT make that protocol correct. While different implementations decrease the number of instances in which CBT is unable to form a complete tree, none eliminate the possibility of a disconnected sub-tree altogether. In addition, the fixes that decrease the number of loops in CBT increase the traffic required to build and repair a multicast tree and increase the latency of nodes attempting to join or rejoin the tree. For this reason, the results from the simulations of OCBT are not included in Figure 5.3; even though OCBT has better performance in repairing the multicast tree following a link failure, it would be unfair to compare this performance to that of CBT, as CBT does not always produce a multicast tree whereas OCBT does.

It is evident that flushing the sub-tree upon detection of a loop reduces but does not eliminate the number of times that a sub-tree is unable to reconnect to the multicast tree. This is completely expected; by flushing the sub-tree routers are able to reconnect to the core directly. There are still times, however, when a core must rejoin the tree and it may be unable to if its path lies through its descendents. Using a core-ping also helps to prevent loops from forming. By sending a unicast core-ping, the protocol avoids building any tree branches while the unicast routing is unstable. Not until any loops between the sender and core are removed will the core-ping be able to reach the core and be answered. This helps avoid building any loops into the tree.

In no case, however, does any CBT implementation examined here eliminate loops completely as OCBT does. In addition, the better performing CBT implementations have a high performance penalty. Core-pings add significantly to the traffic being sent as each branch that is built now adds at least two messages to the total. The join latency for joining routers is also increased by the time it takes for the round trip to the core. Neither of these characteristics are desirable.

Run Number	Build Messages	Repair Messages	Transient Loops	Disconnected Subtrees
1				
CBT-F	72	14.4	9	0
CBT-CP	114	8.0	0	0
CBT-CP,F	114	8.0	0	0
2				
CBT-F	70	5.4	16	1
CBT-CP	109	31.1	0	4
CBT-CP,F	78	5.9	10	0
3				
CBT-F	75	5.4	15	2
CBT-CP	115	8.0	0	1
CBT-CP,F	115	8.0	0	1
4				
CBT-F	75	5.5	18	0
CBT-CP	115	23.8	0	5
CBT-CP,F	115	23.7	0	5
5				
CBT-F	72	17.2	14	3
CBT-CP	110	8.7	0	0
CBT-CP,F	110	8.7	0	0
6				
CBT-F	70	5.1	18	0
CBT-CP	103	14.8	0	2
CBT-CP,F	103	14.8	0	2
7				
CBT-F	71	16	11	2
CBT-CP	112	9.2	0	1
CBT-CP,F	112	9.1	0	1
8				
CBT-F	73	5.6	10	3
CBT-CP	107	23.2	0	4
CBT-CP,F	69	4.9	15	1
9				
CBT-F	73	15.1	11	1
CBT-CP	119	15.5	0	2
CBT-CP,F	120.2	8.7	0	2
10				
CBT-F	72	6.6	22	1
CBT-CP	110	23.8	0	4
CBT-CP,F	112.3	15.9	0	5
11				
CBT-F	71	6.7	16	4
CBT-CP	106	16.2	0	1
CBT-CP,F	108.4	9.6	0	1
12				
CBT-F	68	4.8	11	1
CBT-CP	107	16.0	0	2
CBT-CP,F	108.4	8.0	1	1
Average				
CBT	70	8.7	.25	15.6
CBT-F	71.8	9.0	14.25	1.5
CBT-CP	110.6	16.5	0	2.2
CBT-CP,F	105.4	10.4	2.2	1.6
95% Confidence Interval				
CBT	68.7 - 71.5	4.9 - 12.4	0.0 - 0.5	12.8 - 18.4
CBT-F	70.5 - 73.1	5.8 - 12.2	11.7 - 16.7	0.7 - 2.3
CBT-CP	107.7 - 113.5	11.7 - 21.3	0.0 - 0.0	1.1 - 3.2
CBT-CP,F	95.5 - 115.4	7.1 - 13.8	0.5 - 2.7	.48 - 2.7

Table 5.3: The performance of CBT implementations

6. 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.

As our simulation results show, OCBT eliminates the loops and disconnected sub-trees that occur in the CBT protocol. 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 highlight the importance of a shared tree protocol; OCBT meets that need with correct and efficient performance.

6.1 Future Work

Core placement for the purpose of minimizing packet delay remains an open question for both CBT and OCBT. The flexible nature of core placement in OCBT, however, lends itself to a selection process in which cores could be placed at the convenience of the organization operating the router, subject to the chosen location providing adequate delay and performance. This could lead to a hierarchical scheme for placing cores to limit the scope of the multicast. For example, UCSC could have a single core that would be the connection point to higher-level cores outside the university. A multicast that was not of interest to the outside world could be limited to that core; it would not be forwarded to higher levels.

The major problem with limiting the scope of a multicast using a shared tree is determining which cores to include within the scope. If, in our example, there were two UCSC cores of level one and we desired to limit the multicast to UCSC, we would need to connect these level one routers together. The normal method of doing this would be for each core to connect to the same level two core and limit the multicast at that level. However, if each UCSC router were closer to a different level two router, some mechanism would be required to force them to connect to the same higher level core.

A different solution would be to temporarily promote one level one router to a higher level, as is done for network partitions. This, however, might force a very bad shape to the

multicast tree, resulting in long delays. In addition, this only solves the problem for a single level. As the number of levels increases, the difficulty in forming the tree grows. To ensure that the scope is maintained it may be necessary to designate certain cores as the only connection point to a higher level. All other cores of the same level would connect to that one core, which would carry on the conversation with all higher levels. This exacerbates the problem of delay, and creates a definite hot spot that could be lessened by spreading the load among many routers. Future research will look at this solution to attempt to determine how bad the tree can get while pursuing this solution.

The other research focus will be on using OCBT as a hierarchical protocol linking together regions that are operating some local multicasting protocol. We would like to integrate OCBT with other multicast protocols (e.g. PIM [5], HDVMRP [12], CBT [3]) at a higher or lower level. As a higher-level protocol, a single router in the local multicast group running the local protocol would be designated the OCBT node for the group. That node would then join the OCBT tree and provide connectivity for the local group. Similarly, a single member of a higher-level protocol would also have to be a member of the local OCBT group to provide connectivity. Our work continues to extend OCBT into a multi-layer hierarchical multicast routing protocol in which a multicast routing layer is independent of existing intra-region protocols, much like HDVMRP proposes, but establishing a hierarchical tree structure across regions. This could be difficult as OCBT would not necessarily be able to operate within a particular region; only border routers would be on the OCBT tree. This is a problem with all hierarchical multicast protocols, and is one that is deserving of further study.

Finally, the problem of limiting the scope of core information distribution will be addressed as part of the research outlined above. By limiting core information we can essentially limit the scope of the multicast. In addition, if we can distribute the core information in such a way that we know the boundary routers in a hierarchical scheme, it would simplify the construction of the hierarchical multicast tree.

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.J. Ballardie, P. Francis, and J. Crowcroft, "Core based trees (CBT)", in *Proc.of the ACM SIGCOMM93*, San Francisco, California, Sept. 1993, ACM, pp. 85–95.

[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. E. 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, Sept. 1994, pp. 126–135.

[6] S. Casner, "Are you on the MBone?", *IEEE Multimedia, Summer 94*, pp. 76–79, 94.

[7] A.J. 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.J. Ballardie, "Core based trees (CBT) multicast architecture", Internet Draft I-D, University College London, February 1996, Work in progress.

[9] A.J. 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.

[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.W. 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.

[12] A. S. Thyagarajan and S. E. Deering, "Hierarchical distance-vector multicast routing for the MBone", in *Proceedings of the ACM SIGCOMM95*, Cambridge, Massachusetts, Sept. 1995.