Chapter 3

Appcast - An Application Layer Multicast Protocol

In this chapter, we propose a new Application Layer Multicast Protocol called Appcast. The existing protocols we described in chapter 2 follow two steps to achieve multicast capability. 1. Arrange receivers into an overlay network of uni-cast connections and 2. Construct efficient data distribution trees over this overlay network to distribute data. At the heart of Application Layer Multicast protocols is the overlay topology they create. The topology is created as members/hosts join the multicast group. Application Level Multicast topology building algorithms define a definite relationship among the participating members and thereby create topologies like tree, mesh, hierarchy etc. The relationship can be parent-child, host-neighbors, cluster member – cluster leader etc. While many Application Layer Multicast Protocols are not topology aware as explained in chapter 2, in Appcast we exploit the topology information to better align the overlay topology with the underlying network topology. We also consider exploiting the broadcast nature of the media, which is shielded by TCP/IP. In Appcast, we propose to exploit this broadcast property by re-engineering the TCP behavior, by a method called 'multicast spoofing', which is an extension of ‘TCP Spoofing’. Section 1 of this chapter introduces Application Layer Multicast Protocol. Section 2 describes the basic conditions and method that Appcast follows to create overlay topology and distribute data over it. Section 3 gives algorithms that create Appcast overlay satisfying the conditions laid down in section 2. Section 4 is about optimizing the basic Appcast overlay creation and gives the optimized algorithms. In section 5, we simulate Appcast overlay protocol and other protocols and compare them. Section 6 describes the broadcast media and problems of TCP over such media. Section 7 describes our approach 'multicast spoofing'. Section 8 gives performance improvement results and implementation details of 'multicast spoofing'. Section 9 concludes the chapter.

3.1 Introduction

Each application level multicast protocol differs in how they arrange the hosts into an overlay, manage it and distribute the data over it. To manage the tree, hosts in the overlay keep exchanging information about their health at regular intervals. The overlay
topology considers this overhead of exchanging information as control overhead and every protocol tries to minimize it. The major goal of Application Layer Multicast is to reduce the redundant movements of data on network links through the routers. The number of links a packet crosses to reach all receivers measures whether this goal is achieved by a protocol. The lesser the number of links a packet travels, the more efficient the protocol is in terms of resource utilization like bandwidth, router processing and memory. The overall efficiency of the overlay protocol is measured by two factors – 1. Stress and 2. Stretch. Stress is defined per host or per link as ‘the number of duplicate packets that a host sends or duplicate packets that travel over a link in an overlay’. Stretch, also called as ‘relative delay penalty’ is defined per pair of hosts i.e., source and destination as the ‘ratio between the number of hops a packet travels from source to destination in unicast mode to the number of hops the packet travels in multicast’. In case of IP Multicast, all links experience unit stress and every pair of hosts experiences unit stretch (Chapter 2 gives more details on stress and stretch). Though Application Layer Multicast protocols are not capable of achieving unit stress and unit stretch, they try to balance stress and stretch. Protocols like Bayeux [12] and Scribe [14] are motivated by Peer-to-Peer networks and arrange the hosts into overlay by logically assigning unique number to each host, irrespective of their proximity relations in real underlying network topology. Similarly CAN [18-19] and DTProtocol [11] arrange the hosts by assigning each host a place in a geometric space. The performance of these protocols is heavily taxed by their lack of awareness of underlying topology. The protocol NICE [15-17] is motivated by secure group communication. It first arranges the hosts into a tree topology and groups them into clusters by traversing the entire tree. This grouping is basically to reduce the depth of the tree so that it can contain the control overhead as cluster leaders only exchange information instead of all hosts exchanging information with every one else in the overlay. By doing so, it actually loses the multicast advantage, as communication within a cluster is unicast i.e., larger groups increase the over all links a packet travels to reach all receivers and smaller groups increase the tree depth and hence the control overhead. So an efficient Application Layer Multicast protocols is one, that takes few overall links to disseminate data to all group members, balances stress and stretch and manages the overlay with less control overhead.
3.2 Appcast Overlay Topology

In this section, we describe our proposed 'Appcast protocol' as an Application Layer Multicast Protocol. Appcast creates an overlay topology that is common for all multicast groups. The overlay topology is a tree topology with centralized algorithm to construct the tree. Appcast depends on the topology information to create the tree as it exploits the path redundancy between nodes while creating the tree. One of the nodes, preferably the one owned by the network service provider acts a root node for the multicast tree. Unlike other application layer multicast protocols, wherein every node acts as a routing node also, we clearly demarcate the functionalities of the nodes. In our scheme, we have two kinds of nodes, 1. End hosts and 2. Proxies. Proxies are the ones, who can actually route the applications, where as the end-hosts are the ones which can be either source or destination. In other words, in Appcast tree, proxies can have children, but end-hosts cannot have the children. In all the proposed topology creation algorithms, we considered nodes as proxy nodes only. The end-host nodes information need not proliferate through out the topology and is local to each proxy.

3.2.1 Overlay Topology Creation

Every new proxy that has to join the overlay, contacts the root first. The root determines to which proxy, this new member should get hooked based on the distance metric i.e., number of hops. The algorithms can work for other metrics also, like bandwidth, cost etc. Root selects a proxy (p), as the parent of the new member (m) meeting the following conditions. If such proxy is not found, root itself becomes parent to the new member.

1. \( D(R,m) > D(p,m) \)
   
   The distance between root and new member is greater than the distance between proxy and new member.

2. For all proxies \( P_i \) that satisfy condition 1, \( D(p,m) \leq D(P_i,m) \)
   
   There can be more than one proxy satisfying the condition 1, and if so, choose the ones which are nearer to the new member.

3. \( \text{Path(Parent(p),m)} \) contains \( \text{Path(p,m)} \)
   
   The path between proxy and member is a prefix to the path between proxy’s parent and member. The path need not be shortest path.
When a **new** member is about to join the group satisfying the above conditions will create three cases. 1. A new member's arrival does **not** affect the overlay structure. 2. A new member takes over its parent by reordering the parent-child relationship. 3. A new member takes over its siblings as their parent.

**Case1: A new member's arrival does not affect the overlay structure**

In this case the new joining proxy $P_n$ gets a $P$, which is very near to it. Introducing this proxy into overlay does not affect rest of the tree structure. Figures 3.1a and 3.1b depict this scenario. The figures 3.1a and 3.1b show a network with 8 routers and 3 proxies, in which $S$ acts as the root node and $P1$ and $P2$ joined the overlay in that order. In this scenario, the algorithm selects $S$ (root) as the parent of $P1$ and when $P2$ also selects $S$ as its parent, the overlay relation of $S$ and $P1$ remain unchanged.

![Figure 3.1a Appcast Topology - Case1](image1)

![Figure 3.1b Appcast Overlay](image2)

**Case2: A new member takes over its parent by reordering parent-child relationship**

The new joining proxy $P_n$ in this case selects a $P$ such that $P_n$ is in the path of $P$'s parent and $P$. In this case as in figure 3.2 series, the algorithm first selects $P1$ as the parent of $P3$ as per condition 1, but to meet the condition 3, it has to reorder the parent-child relationship, i.e., $P3$ becomes parent of $P1$ and child to $S$. 
Case3: A new member takes over its siblings as their parent
In this case, the new proxy $P_n$ selects a parent $P$ such that $P_n$ is in the path of $P$ and some of the children proxies of $P$. In figure 3.3 series, $P_4$ selects $S$ as its parent and since $P_4$ is the child of $S$ and is in the path of $S$ and $P_3$, $P_4$ takes over as parent to $P_3$. 

Figure 3. 3a: Appcast Topology: Case3  Figure 3.3b: Before Taking over sibling  Figure 3.3c: After Taking over Sibling
3.2.2 Appcast Topology Creation - A Low Stress and High Stretch Overlay Protocol

In this section, we describe the algorithms that create the overlay topology satisfying the conditions and cases discussed in section 3.2.1. Every new proxy willing to join the group sends a join message to the root. The root invokes the function “FindNearestProxy” which returns a proxy that is closest to the node. Root then calls “FindRelations” to fix the relationship of the new joining proxy and others in the overlay tree satisfying the conditions laid in section 3.2.1.

The well-known Dijkstra’s algorithm [117] finds the shortest paths from a source to a destination (vertices) or all destinations in a graph. Dijkstra’s algorithm keeps two sets of vertices. 1. The set of finished vertices and 2. Set of active vertices (active set). A vertex is active if it has a temporary label and a finite distance but the investigation of the vertex have not yet been finished. The algorithm terminates once the required destination joins the first set, in case it has to find path between source and destination or once the 2nd set becomes null, in case it has to find paths between source and all destinations.

**Dijkstra’s Algorithm**

Let V be set of n vertices in graph G representing the given network. S is the set of vertices whose labels are permanent. Let V-S be the set of active vertices with temporary labels. C[i,j] is the cost of edge between vertex i and vertex j. D[i] is the distance from source node to the destination node i. Label[i] denotes the predecessor node to reach node i. The pseudo code algorithm for finding shortest paths from a single source v₀ to all nodes in a network represented by graph G is as given below.

```
Dijkstra (graph G, node v₀) {
    S={v₀}
    For i = 1 to n {
        D[i] = C[v₀,i]
        For w in V-S with min D[w]
            Add w to S
            For each node v in V-S with edge to w
                If (D[v] < D[w] + C[w,v]) {
                    D[v] = D[w] + C[w,v]
                    Label[v] = w
                }
    }
}
```
**Complexity of Dijkstra's Algorithm**

The complexity of Dijkstra's algorithm \([118]\) is based on two operations: 1. Finding a node with minimum distance. This has a complexity of \(O(N)\) and 2. Changing the labels with complexity \(O(m)\). These two steps are performed \(N\) times and so the total complexity of the algorithm is \(O(N^2)+O(mN)\).

**Appcast - FindNearestProxy Algorithm**

This algorithm is written by modifying the above Dijkstra's Algorithm. The base Dijkstra's algorithm finds shortest path between a source and destination. The algorithm starts from the source node and keeps fixing the labels for intermediary nodes to reach the destination node and terminates the algorithm once it reaches the destination. In our case, the source node is the 'joining proxy' and destination node is the 'root'. The problem of finding the nearest proxy turns out to be finding an intermediary node (this node has to be proxy) in Dijkstra's algorithm while traversing from source to destination via shortest path i.e., 'joining proxy' to 'root'. Apart from the two sets that Dijkstra's algorithm keeps; we keep one more set of nodes i.e., set of all proxies. We change the algorithm such that the algorithm terminates once it reaches any node that belongs to this set. The pseudo code for this algorithm is given below.
Complexity of FindNearestProxy Algorithm

As said earlier, the complexity of Dijkstra's algorithm is based on two operations:
1. Finding a node with minimum distance. This has a complexity of $O(N)$ and 2. Changing the labels with complexity $O(m)$. These two steps are performed $N$ times and so the total complexity of the algorithm is $O(N^2)+O(mN)$. In addition to this, in our algorithm, we have to find whether the selected node with minimum index belongs to the set of proxies. This has a complexity of $O(k)$. These three steps are performed $N$ times and total complexity is $O(N^2)+O(mN)+O(kN)$.

3.2.3 Appcast Topology Management

In this section, we describe how Appcast, manages the overlay that it created keeping the relationships of members intact, i.e., satisfying the conditions laid down in section 3.2.1. The topology management has to take care of members leaving and links down, members down etc., scenarios. The root keeps information of all proxies. Every proxy keeps information about its parent and its children. Also, every proxy keeps track (heart beat) of its children and parent. If any proxy is down, immediately, its children contact the root and try to hook to the parent of the downed proxy. It is the root, which tells the children about their new parent, keeping all constraints satisfied. Whenever, a new proxy joins or leaves, few other proxies also will be informed by root to change their relationships, so that the constraints are satisfied.
Appcast - FindRelations Algorithm

Given below is the algorithm that takes care of all three cases whenever a new proxy joins the overlay. This algorithm is invoked by ‘root’ after the algorithm ‘FindNearestProxy’.

```
FindRelations(node V_o, proxy NP)
/* This algorithm rearranges the overlay, as a new proxy V_o joins the overlay,
selecting NP as its parent */
{
    if path(NP.Parent, NP) contains V_o /* Case 2 */
        NP=NP.Parent
        \begin{itemize}
        \item 1. Root changes the NP as NP’s Parent as V_o is in the path between NP.Parent and NP. Now, old NP will be in the path of this new NP and V_o, and so follows case 3
        \end{itemize}
    \end{itemize}
    For all children ch in children(NP) /* Case 3 */
    {
        if (path(NP,ch) contains V_o
            {
            remove ch from children(NP)
            add ch to children(V_o)
            set ch.parent=V_o
            }
        }
    }
    Set V_o.parent=NP /* Case 1 */
    Add V_o to children(NP)
    \begin{itemize}
    \item Simple case1, where node V_o is just added NP as child.
    \end{itemize}
```

Appcast Data Movement

In Appcast, data can be flowed bottom-up and top-down across the Appcast topology tree. To avoid loops, each node checks from which it received the data and accordingly forwards to selective children and parent. Whenever a proxy receives data, it checks from whom it received. If it received from its parent, then it forwards packets to all its children. If it received packet from one of its children (i.e., not parent), it forwards the packet to its own parent and to all its children except the child from whom it received.
Appcast - MulticastForward Algorithm

Every proxy in the overlay calls the following algorithm as and when it receives a piece of information on overlay.

```plaintext
MulticastForward(node sender, node receiver) {
    if receiver.parent<>sender {
        MulticastForward(receiver, receiver.parent)
    }
    for each receiver.child {
        if receiver.child<>sender {
            MulticastForward(receiver, receiver.child)
        }
    }
}
```

1. If the current node received data from child, proliferate the data upwards the tree i.e., Send data to
   For all children of the current node proliferate the data i.e.,
   Distribute the data downwards the tree. Avoid sending the data to the sender itself.

3.3 Appcast Algorithms

In this section, we give all the algorithms described above as a series. Following are the notations used in the Appcast Algorithms.

\[
\begin{align*}
G & \quad \text{connected graph representing the network} \\
v_0 & \quad \text{is the proxy joining to multicast group} \\
r & \quad \text{root node} \\
V & \quad \text{is the set of all vertices in the graph } G \\
S & \quad \text{set of all vertices with permanent labels} \\
P & \quad \text{Set of all proxy nodes on } G \\
n & \quad \text{number of vertices in } G \\
D & \quad \text{set of distances to } v_0 \\
C & \quad \text{set of edges in } G \\
NP & \quad \text{Nearest Proxy found from the algorithm “FindNearestProxy”} \\
\text{Distance}(w, v_0) & \quad \text{is the unicast distance from node } w \text{ to node } v_0 \\
\text{C_Distance}(w, r) & \quad \text{is the cumulative (application level) distance from node } w \text{ to } r \\
\text{Accept_children} (w) & \quad \text{is the number of children node } w \text{ can accept} \\
\text{Cur_children}(w) & \quad \text{is the number of children node } w \text{ has} \\
\text{D} & \quad \text{Set of Distributing Proxies} \\
O & \quad \text{Set of Publisher Proxies}
\end{align*}
\]
Algorithm 1:
1. NearestProxy FindNearestProxy (graph G, node v₀, node r, Proxies P)
   /* Returns the 'NearestProxy' from the set of proxies 'P', to which 'v₀' can be a child. If no such proxy is found, v₀ is attached to root r */
   {
      S={v₀}
      For i = 1 to n
         D[i] = C[v₀, i]
      For i = 1 to n-1
         {
            Choose node w in (V-S) with min D[w]
            If node w belongs to P
               {
                  add v₀ to P
                  return w
               }
            Add w to S
            For each node v in (V-S)
               D[v] = min(D[v], D[w] + C[w,v])
         }
   }

Algorithm 2:
2. FindRelations(node V₀, proxy NP)
   /* This algorithm rearranges the overlay, as a new proxy V₀ joins the overlay, selecting NP as its parent */
   {
      if path(NP,Parent, NP) contains V₀ /** Case 2 ***/
         NP=NP.Parent
      For all children ch in children(NP) /** Case 3 **/
         {
            if (path(NP,ch) contains V₀
               {
                  remove ch from children(NP)
                  add ch to children(V₀)
                  set ch.parent=V₀
               }
         }
      /***** Case 1 *****/
      Set V₀.parent=NP
      Add V₀ to children(NP)
   }
3.4 Appcast Optimization - A Balanced Multicast Overlay Protocol

In this section, we propose optimization to Appcast overlay creation and data distribution by using two approaches. In the first approach, the performance criteria of any Application Layer Multicast protocol like Stress and Stretch have been considered and the protocol tries to optimize these criteria. In second approach, a publisher and distributor approach is considered wherein a publisher is the one who can own and originate and distribute the data, while the distributor is the one who just can keep distributing the data without any right to originate the data. We describe both the approaches in the following sections.

3.4.1 Appcast Optimization using Stress and Stretch Criteria

In Appcast described in section 3.3, a proxy joining the multicast group selects the very first proxy that it comes across while finding the path from itself to the root. This approach definitely ensures that the path length from the 'joining proxy' to the 'parent proxy' is lesser than the path length from the ‘joining proxy’ to the ‘root’. However, if we take into consideration the actual path length from root to this new proxy (along the proxies), the path would be lengthier.

Algorithm 3:

```c
3. MulticastForward(node sender, node receiver)

/****** A recursive algorithm called by every proxy in the overlay to distribute
data over the Appcast Overlay *********/
{
    if receiver.parent<>sender /**Distribute data upwards ***/
    {
        MulticastForward(receiver,receiver.parent)
    }
    for each receiver.child /** Distribute data downwards ***/
    {
        if receiver.child<>sender /** Avoid loops by not sending data
to the sender again ***/
        {
            MulticastForward(receiver, receiver.child)
        }
    }
}
```
The performance results as in section 5, clearly show that Appcast uses very few over all links/hops. At the same time, it also show the maximum application level path lengths i.e., maximum stretch. To keep the stretch and stress at an optimum level, the Appcast_optl algorithm is proposed. In this algorithm, a ‘joining proxy’ can specify how many children (stress) it can accept and how much stretch (delay) it can bear. So the algorithms described in section 3.2 are modified to accommodate the above optimization for ‘FindNearestProxy’ and ‘FindRelations’ algorithms, while the ‘Multicast Forward’ algorithm remains same. Both the algorithms are modified to verify the ‘stress’ capacity of the ‘proxy’ chosen by a joining node and ‘stretch’ criteria chosen by the joining node itself. The algorithm ‘MulticastForward’ does not require any modifications, as all proxies are arranged into an overlay only after verifying their capacities and so at the time of distributing the data, they need not be verified for the capacity.

Algorithm - FindNearestProxy_Optl

FindNearestProxy_Optl(graph G, node v₀, node r, float acpt_stretch)

/* This algorithm finds a ‘NearestProxy’ that satisfies the stretch criteria imposed by node v₀, and the stress criteria imposed by ‘NearestProxy’ */

for i = 1 to n-1 {
    \( D[i] = C[v₀, i] \)
}

for i = 1 to n-1 {
    if node w belongs to P {
        Even if a proxy is found, it is verified for its capacity.
        if \( ((\text{Distance}(w, v₀) + C \cdot \text{Distance}(w, r)) / \text{Distance}(v₀, r) < \text{acpt_stretch}) \&\& \text{acpt_children}(w) > \text{cur_children}(w)) \) {
            add \( v₀ \) to P
            \( \) return \( w \) \)
        }
    }
    if node w = r, return r
    Add w to S
    for each node v in V - S
    \( D[v] = \min(D[v], D[w] + C[w, v]) \)
}

3.4.2 Appcast optimization using Publisher and Distributor proxies

In addition to the optimization given using the criteria like stress and stretch specified by each joining proxy, even this algorithm is modified from FindRelations algorithm of section 3.2.

```
Algorithm FindRelations_opt1(node V₀, node NP)
Following the optimization criteria set by each joining proxy, even this algorithm is modified from FindRelations algorithm of section 3.2.

FindRelations_opt1(node V₀, node NP) {
    if path(NP.Parent, NP) contains V₀, /* Case 2 */
        NP=NP.Parent
    for all children ch in children(NP) { /* Case 3 */
        if (path(NP,ch) contains V₀ & & Acpt_children(V₀) > Cur_children(V₀)){
            remove ch from children(NP)
            add ch to children(V₀)
            set ch.parent=V₀ }
    /* Case 1 */
    Set V₀.parent=NP
    Add V₀ to children(NP)
    Add V₀ to P
}
```

Relationship is modified only if the node $V_0$ has capacity.

Capacity need not be verified as its already taken care in FindNearestProxy_Opt1 algorithm in this case.

Stretch need not be verified, as it is always less from NP.Parent to $V_0$ compared to NP to $V_0$

3.4.2 Appcast optimization using Publisher and Distributor proxies

In addition to the optimization given using the criteria like stress and stretch specified by each proxy, a ‘proxy’ can also specify whether it is a distributor or a publisher at the time of joining. A distributor is the one who can only distribute the data and a publisher is the one who can originate as well as distribute data. The algorithm ensures that all publishers are nearby and can proliferate the data faster onto every node of the tree. Otherwise, if the publisher happens to be at the end of the tree, the data originated from it will take long time to pass over all the nodes of the tree. Also, a distributor cannot send the data to a publisher, whereas publishers can send data to any one, including other publishers and distributors. Opt2 series of algorithms use this concept.
NearestProxy FindNearestProxy_opt2(graph G, node V₀, node r, float acpt_stretch) {
    S = {V₀}
    if V₀ is a publisher {
        Add V₀ to O
        P = O
    }
    else {
        Add v₀ to D
        P = O + D
    }
    for i = 1 to n
    D[i] = C[v₀,i]

    for i = 1 to n-1
    {
        Choose node w in (V-S) with min D[w]
        if node w belongs to P
        {
            if (((Distance(w,v₀)+ C_Distance(w,r))/ Distance(v₀,r) < acpt_stretch) &&
                acpt_children(w)>cur_children(w))
            return w
        }
        if node w = r
            return r
        Add w to S
        for each node v in V-S
            D[v] = min(D[v], D[w] + C[w,v])
    }
}
Data Distribution

Data can be flowed bottom-up and top-down across the Appcast topology tree. Whenever a proxy receives data, it checks from whom it received. If it received from its parent, then it forwards packets to all its children. If it received packet from one of its children (i.e., not parent), it forwards the packet to its own parent and to all its children except the child from whom it received. Since we differentiate proxies as distributors and publishers, if receiver proxy is not a publisher, it cannot send data to its parent. Distributors can distribute the data downwards, not upwards, as a distributor proxy can never originate the data. With these differences the algorithm ‘MulticastForward’ is changed as ‘MulticastForward_opt2’ algorithm.
MulticastForward_opt2(node sender, node receiver) {
    if receiver.parent<>sender && receiver is a Publisher
    {
        MulticastForward(receiver, receiver.parent)
    }
    for each receiver.child {
        if receiver.child<>sender {
            MulticastForward(receiver, receiver.child)
        }
    }
}

3.4.3 Appcast Optimization Algorithms
In this section we give all algorithms described in section 3.4.2 as a series of algorithms.

Algorithm 4:
4. NearestProxy FindNearestProxy_opti(graph G, node v0, node r, float acpt_stretch)
/* This algorithm finds a 'NearestProxy' that satisfies the stretch criteria
imposed by node v0, and the stress criteria imposed by 'NearestProxy' */
{
    S={v0}
    for i = 1 to n
        D[i] = C[v0,i]
    for i = 1 to n-1
    {
        Choose node w in V-S with min D[w]
        if node w belongs to P
        {
            if \[\left(\frac{\text{Distance}(w,v0) + \text{C}_{\text{Distance}}(w,r)}{\text{Distance}(v0,r)} \right) < \text{acpt_stretch} \&\& \text{acpt_children}(w) > \text{cur_children}(w)\] \n            {
                add v0 to P
                return w
            }
        }
        if node w = r, return r
        Add w to S
        for each node v in V-S
        D[v] = min(D[v], D[w] + C[w,v])
    }
}
Algorithm 5:
5. FindRelations_opt1(node V0, node NP)
{
    if path(NP.Parent, NP) contains V0 /** Case 2 **/
    NP=NP.Parent
    for all children ch in children(NP) { /** Case 3 **/
        if (path(NP, ch) contains V0 && Acpt_children(V0) > Cur_children(V0)) {
            remove ch from children(NP)
            add ch to children(V0)
            set ch.parent=V0
        }
    }
    /** Case 1 **/
    Set V0.parent=NP
    Add V0 to children(NP)
    Add V0 to P
}

Algorithm 6:
6. NearestProxy FindNearestProxy_opt2(graph G, node v0, node r, float acpt_stretch)
{ S={v0}
  if v0 is a sender {
    Add v0 to O
    P=O
  } else {
    Add v0 to D
    P=O+D
  }
  for i = 1 to n
    D[i] = C[v0,i]
  for i = 1 to n-1 {
    Choose node w in (V-S) with min D[w]
    if node w belongs to P {
      if ((Distance(w,v0) + C_Distance(w,r)) / Distance(v0,r) < acpt_stretch) && acpt_children(w)>cur_children(w))
        return w
      if node w = r
        return r
      Add w to S
      for each node v in V-S
        D[v] = min(D[v], D[w] + C[w,v])
    }
}
3.5 Comparative study

The protocols (CAN [18-19], Bayeux [12], DTProtocol [11] etc.,) that have no knowledge of the underlying topology suffer poor performance and can help only in sharing and distributing the load of the source across the members. The mesh based protocols like ESM [1-2] and Yoid [20-21] suffer from control overhead $O(N^2)$ and are not suitable for large groups. The tree and hierarchical topologies like HMTP [13], TAG...
[22] and NICE [15–17] are able to contain the control overhead and at the same time performing well. Following table shows the intuitive comparison metrics.

Table 3.1 Application Layer Multicast Protocol Comparison

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Max Path Length</th>
<th>Max Tree Degree</th>
<th>Avg. Control Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protocol</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Appcast</td>
<td>Parameterized</td>
<td>Parameterized</td>
<td>O (Max. Degree)</td>
</tr>
<tr>
<td>ESM</td>
<td>Unbounded</td>
<td>Unbounded</td>
<td>O (N^2)</td>
</tr>
<tr>
<td>Void</td>
<td>Unbounded</td>
<td>O (Max. degree)</td>
<td>O (Max. degree)</td>
</tr>
<tr>
<td>HMTP</td>
<td>Unbounded</td>
<td>O (Max. degree)</td>
<td>O (Max. degree)</td>
</tr>
<tr>
<td>CAN</td>
<td>O (dN^{1/3})</td>
<td>Constant</td>
<td>O (log N)</td>
</tr>
<tr>
<td>Bayeux</td>
<td>O (log N)</td>
<td>O (log N)</td>
<td>O (log N)</td>
</tr>
<tr>
<td>TAG</td>
<td>Unbounded</td>
<td>Unbounded</td>
<td>O (Max. tree degree)</td>
</tr>
<tr>
<td>NICE</td>
<td>O (log N)</td>
<td>Constant</td>
<td>Constant</td>
</tr>
</tbody>
</table>

3.5.1 Simulation and results

For comparison purposes we considered only TAG [22], NICE [15-17] and HMTP [13] as our proposed protocol Appcast also is a tree based protocol. The figures 3.5, 3.6 and 3.7 show the overlay topologies created by them, when taken the network shown in figure 3.4. Figure 3.8 shows the overlay created by Appcast. R1,R2,R3,...R10 represent routers and S,A1,A2...A5 nodes in figure 3.4. The order of joins of the nodes is A3, A4, A5, A1 and A2. Since TAG chooses its parent based on the longest path match over shortest path from node to root, A3, selects A2 as parent, though A1 is nearer to it. Order of joins, matter a lot for the performance of HMTP. Since 'A1' joined in the last, it just took the one as its parent, which is nearer to it ie A3, without checking whether it is on the way between S and A3. NICE groups nearby members into clusters and arranges these clusters into a hierarchy.
We used Boston University's Network Topology generator - BRITE [77] to simulate our experiments. BRITE generates different kinds of network topologies based on the models - Flat Router Level models (Router Waxman, Router Barbasi-Albert); Flat AS Level models (AS-Waxman, AS-Barbasi-Albert) and Hierarchical models (Transit-stub, tiers). First, we generated 100 nodes in AS model and assigned 20 hosts to these nodes. In this experiment, HMTP [13] showed higher stress and lower stretch. TAG [22] showed even more higher stress and less/no stretch. NICE, with cluster members fixed to 3, almost showed similar result like HMTP. Similar experiments have been conducted on network topology with 1000 nodes and with varying group memberships of hosts. Figure 3.9 shows hop comparison, figure 3.10 shows application level hops comparison; figure 3.11 shows stretch comparison and figure 3.12 shows the stress comparison of unicast, NICE, Appcast, HMTP and TAG.
Appcast used overall less hops followed by HMTP and TAG used almost similar hops like unicast. This is because - TAG node will select a parent, which has maximum overlapping shortest path with it. In other words, TAG does not look into alternative paths. This makes most of the nodes select the source itself as their parent i.e., very few nodes get nodes other than source as parent.

Figure 3. 9: Hops Comparison

Figure 3. 10: Application Level Hops Comparison
TAG showed application level hops almost similar to unicast, as TAG does not look into alternative paths. NICE [15-17], while showing less over all hops compared to TAG, showed the higher application level hops compared to TAG and HMTP. This is because, with in clusters, NICE uses normal unicast among the cluster members and clusters leaders. As the group size increases, application level hops increase tremendously for NICE. Appcast is the one, which used the less number of hops. However, it is the one, which used maximum application level hops. This is because; it does not use any mechanism to control the tree depth. For this reason, an optimized version of Appcast protocol has been proposed, in which each joining host can specify the stretch parameter - the ratio between unicast hops and application level hops. As far as stretch is concerned, TAG showed less stretch and Appcast showed high stretch. NICE showed less stress and TAG showed high stress.

Figure 3.11: Stretch Comparison

![Stretch Comparison Graph]

53
3.6 TCP/IP Over Satellite

The paper [53] described issues and pitfalls of satellite-based systems. Very Small Aperture Terminal (VSAT) networks are well known for delivering broadcast mode traffic. By nature broadcasting via satellite is simpler than for terrestrial networks, which are point-to-point networks. The wide range of content delivery supported by satellite clearly illustrates the versatility of this medium. The delivery of high-speed and interactive data via satellite has its own unique problems [53]. Geo-stationary satellite systems have a fixed round trip delay of 600ms that cannot be avoided. TCP/IP allows certain amount of data limited by window size, without an immediate acknowledgement from the receive end [57]. For a satellite link with round-trip delay of 0.8 seconds and bandwidth capacity of 1.54 Mbps, the theoretical optimal window size is 154 Kbytes, which is far higher than the maximum allowed TCP window size of 64K. TCP requires a three-way handshake between sender and receiver, before actual data start passing the network [55] i.e., three control packets should be exchanged between sender and receiver. For a satellite network with 600ms round-trip time, 1.8 sec is required for every communication session before the first data packet is exchanged.

To maximize bandwidth utilization in a satellite network, TCP needs a much larger window size. A new TCP extension, or TCP-LW for "large-window" [59], has been defined to increase the maximum window size from $2^{16}$ to $2^{32}$, allowing better utilization of links with large bandwidth delay products (bandwidth x delay). To obtain
good TCP performance over satellite links, both sender and receiver use a version of TCP that implements TCP-LW. Transaction TCP, or T/TCP [55,63], is an extension to TCP designed to make 3-way handshake more efficient. T/TCP does this by bypassing the three-way handshake and slow start, using the cached state information from previous connections. Although T/TCP is designed mainly for short client-server interaction applications, it can be used to reduce the impact of latency on the beginning of a TCP connection.

TCP spoofing is another mechanism devised to increase the satellite performance using TCP/IP. In TCP spoofing as shown in figure 3.13, an intermediate gateway (usually at the satellite uplink) prematurely acknowledges a TCP segment without waiting for the actual acknowledgment from the receiver [53]. This gives the sender the illusion of a low-latency network so the TCP slow start phase can progress more rapidly. The intermediate gateway buffers segments in transit. When the actual acknowledgment from the receiver arrives at the gateway, it is suppressed to prevent duplicate acknowledgments from reaching the sender. When receiver’s acknowledgment never arrives and the gateway times out, it retransmits the lost segment from its local buffer.

Figure 3.13: TCP Spoofing
3.7 Multicast Spoofing

Satellite communication has a property of more inward capacity and less upward capacity at remotes coupled with high latency, i.e. a VSAT can receive more data than it can send. Sometimes even sending an acknowledgment can prove costly. Since Satellite communication is reliable, a mechanism called "TCP Spoofing" [55-59] (also explained in section 2.2.3.2) has been devised such that the sender keeps sending data without waiting for the actual acknowledgments arrive.

In TCP spoofing, an intermediate gateway (usually at the satellite uplink) prematurey acknowledges a TCP segment without waiting for the actual acknowledgment from the receiver. This gives the sender the illusion of a low-latency network so that TCP slow start phase can progress more rapidly. The intermediate gateway buffers segments in transit. When the actual acknowledgment from the receiver arrives at the gateway, it is suppressed to prevent duplicate acknowledgments from reaching the sender. We extend the 'TCP spoofing' behavior to the needs of multicasting as described in next paragraph.

The Approach

Applications that use IP Multicast assign a common IP multicast address and port number at which all clients (receivers) keep listening for the sender (server) to send information. This requires client application software to be aware of the multicast capability and write the code specifically with multicast API i.e., the software on clients in multicast mode and on clients in unicast mode has to be different. Also, many application developers are not aware of the multicast techniques (a network specialization) and seldom show any interest to develop such applications. Commercial high-speed multicast applications confined to general applications like audio and video distribution and are controlled mainly by satellite communication vendors like Hughes.

Our aim is to develop a methodology that simplifies the multicast application development exploiting the broadcast nature of the media. In next chapter we show exact applications like a) ‘file push service’ which pushes a static file to all recipients, b) 'database replication service' which pushes a XML file that contains both data and events to recipient databases for the databases to take action against the data sent based on the events and c) 'auction service' which pushes the bid quotes to all recipients in broadcast mode and receives responses in unicast mode. Information is pushed on to the recipients
in either multicast or unicast mode based on the media through which recipients are connected to sender. Recipients need not use separate software, however they inform the sender of their media. All these applications are designed on standard Internet Protocols like HTTP, SMTP etc with SOAP [47-49] as wire protocol.

We modified the concept of "TCP Spoofing" to suit our requirement such that the gateway behavior is implemented at sender and receiver. We allocate a multicast address common to all the receivers. We implement multicast over normal TCP using "TCP Spoofing". The sender keeps sending information to the multicast address in unicast mode only. At the sender, the gateway itself generates acknowledgement packets with the source IP address of the receiver, without actually receiving them over wire as shown in figure 3.14.

![Diagram of multicast spoofing](image)

*Figure 3.14: Multicast Spoofing*
At the receiver, the gateway rewrites (spoofs) the destination multicast address as the receiver's address before handing it over to the TCP layer. It also suppresses the acknowledgements. This way the sender keeps on pumping data packets and it's assumed that all packets reach the receiver. In future this can be modified, such that one of the receivers only acknowledges the packets, to ensure that all packets reach the receivers.

The steps required at the sender side and client side to receive data in multicast mode are as given below.

**Server:** At server the following events take place to send data to receivers.
- The sending application sends the information at URL http:\service.com\filepush.
- The URL is converted to an IP address whose incoming packets can be received by all recipients connected through the broadcast media.
- The TCP stack at sender sends the packets as usual.
- The gateway software generates a spoofed acknowledgement of the packet sent and sends back to the TCP stack.
- TCP stack after receiving the acknowledgement releases the next packet.
- The above process is repeated till all the packets are sent.

**Receiver:** At receiver the following events take place to receive data.
- Receiver sets up a web service at the URL http:\service.com\filepush on its own machine. This webservice can receive the information sent by the server.
- Receiver gateway rewrites the packets destination IP address with its own IP address and hands over the packet to the TCP/IP stack of the receiver.
- The service at receiver behaves as if the information has arrived in unicast mode.
- The TCP/IP stack at receiver replies with an acknowledgement to the packets it received.
- The gateway software at receiver suppresses those acknowledgments.

Both sender and receiver follow the above steps. While the gateway software takes care of rewriting the packets, generating and suppressing the acknowledgments; the
application software is just not aware of the fact that they are in multicast mode i.e., application developers need not worry about the network specialization.

With the above multicast spoofing method available at proxies connected through broadcast medium, our proposed Appcast protocol creates topologies with both unicast and broadcast links as shown in figure 3.15.

Figure 3. 15: Appcast overlay with unicast and broadcast links

3.8 Simulation and Performance Results of 'Multicast Spoofing'

We simulated the satellite environment on an Ethernet LAN with 3 computers out of which one acted as sender and two as receivers. We used ndis3NT software written by Dan Lanciani, on windows 2000, which allows altering the packets before and after TCP/IP stack. Below we describe the crucial API provided by this software.

1) **nd_send_to_tcp(int unit, char *data, int len, int hlen)**
   This function sends a frame to MSTCP just as if it had been received from the network. The len is the total length of the frame and hlen is the hardware header portion (e.g., 14 for Ethernet and emulations thereof)-

2) **nd_send_as_tcp(int unit, char *data, int len)**
   This function is identical to nd_send_pkt except that ndis3pkt's tcp/ip multiplexor treats the frame as if it had been generated by MSTCP.

The **ACCESS_FLAG_TUNNEL** flag is applied to nd_access_type in the typelen argument. It indicates that the handle is to receive any packets that MSTCP sends on the interface. Only one handle on each interface can be opened in TUNNEL mode. As long
as such a handle is open, MSTCP’s normal reception of packets is blocked (i.e., it receives packets only from nd_send_to_tcp) and all packets that it sends are intercepted and rerouted to the handle. When the handle is released, MSTCP operation returns to normal.

The ACCESSFLAGASMSTCP flag is applied to nd_access_type in the typelen argument. It indicates that, for purposes of ndis3pkt's tcp/ip multiplexor, received frames are treated as if they were destined for MSTCP.

A typical Win32 intermediate filter would create two handles, one with ACCESS_FLAG_TUNNEL and one with ACCESS_FLAG_ASMSTCP. It would read from the first handle, apply any desired changes to the output packets, and then forward them to the network with nd_sendastcp. It would read from the second handle, apply desired changes to the input packets, and then forward them to MSCTP with nd_send_to_tcp. Entire packets can, of course, be added or deleted from the stream.

We kept the delay as 250ms for regular sender to receive the acknowledgement, just to emulate the satellite environment i.e., receiver sends acknowledgement only after 250ms of receiving a packet from sender. We calculated throughput (bytes transferred per sec) in two cases - 1) with regular sender and 2) with spoofed sender. At the time of spoofing, we generated spoofed acknowledgements at sender and suppressed the actual acknowledgements at receiver itself. As the spoofing is at TCP stack level, packets can be sent transparently by any application. We sent packets of size 1500bytes using FTP protocol. We calculated the actual throughput, when we sent 200, 400 up to 2000 packets. The graph shows that throughput is less when few packets have been sent and is stabilized as number of packets grew in case of ‘Regular Sender’. This is due to TCP’s 3-way handshake mechanism. But in case of ‘Spoofed Sender’, throughput remained constant almost.
3.9 Conclusions
The proposed application level multicast protocols basically differ in the overlay
topology creation and distribution of data over the same. While studying the existing
protocols, it has been found that mesh based systems are complex to maintain and tree
based systems give good performance and less control overhead. In both the tree-based
systems i.e. TAG [22] and HMTP [13], new joining node traverses the tree from root,
down the children. While TAG is relying on shortest path, HMTP relies on shortest
distance. These features some times may lead to overlapping links. We proposed a new
method that allows the joining node to select a parent, which is on its way to the source.

This chapter also described about problems faced using TCP in a satellite
environment. TCP indeed deteriorates the actual capacity of a satellite link because of its
window size, 3-way handshake etc. Failure of TCP over high bandwidth capacity with
long latency clearly shows that TCP was not designed for a reliable and high-speed
network. We found that spoofing is indeed beneficial for large file transfers. Spoofing's
benefit to web servers and other content providers may be significant. In chapter 4, we
show how we designed applications on Appcast that can cater information both in unicast
and multicast mode, without any special effort from the developer to develop multicast
applications. On the proposed new topology-building algorithm, we use SOAP [47-49] as application level transport mechanism and implement applications like Mass Information Push, Database Replication and E-Auctions. Chapter 4 gives details of these applications and application architecture with comparison to other architectures.