AN ABSTRACT OF THE THESIS OF

Krishnan Kolazhi for the degree of Master of Science in Computer Science presented on June 5, 2006.

Title: Node and Topology Management for Content Distribution in Source Constraint Networks

Abstract approved: _____

Thinh Nguyen

As broadband Internet becomes widely available, Peer-to-Peer (P2P) applications over the Internet are becoming increasingly popular. Such an example is a video multicast application in which, one source streams a video to a large number of destination nodes through an overlay multicast tree consisting of peers. These overlay multicast-based applications, however, do not exploit the full bandwidth of every peer as the leaf nodes in the overlay multicast tree do not contribute their bandwidth to the system. On the other hand, all the peers in a properly constructed overlay mesh can contribute their bandwidth, resulting in high overall system throughput. This thesis provides details of an overlay topology that optimizes the bandwidth usage and also discusses design issues in node and topology management. The thesis also presents implementation details of a real world P2P system based on the above mentioned topology. Finally the designed system is deployed on machines across PlanetLab and the results are presented. Large scale simulation results are also presented to verify robustness of the system.

Node and Topology Management for Content Distribution in Source Constraint Networks

by

Krishnan Kolazhi

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented June 5, 2006
Commencement June 2007

Master of Science thesis of Krishnan Kolazhi presented on June 5, 2006

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Krishnan Kolazhi, Author

ACKNOWLEDGMENTS

I would begin by thanking my advisor Dr. Thinh Nguyen without whose support and ideas this thesis would not have been possible. Also I would like to thank my project group members Rohit Kamath and Phuoc Do for their co-operation and suggestions at times when I needed them the most.

I would like to thank the Information Services Department, for keeping me funded for the 2 years of my studies and for being accommodating so that I could focus on my studies.

I would like to extend my gratitude to my labmate Sunand Tullimalli for taking time off his research and helping us with PlanetLab experiments.

I would like to thank Dr. Michael Quinn, Dr. Tim Budd, Dr. Toshimi Minoura and Dr. Rajeev Pandey whose courses and ideas have influenced me and hence this project, a lot.

I would like to thank my family back in India for their constant support and encouragement. Credit also goes to my cousin who has guided me thorough tough times at the start of my course. He has always been there when I needed any help or guidance. Also, I would like to thank my roommates and friends for making these 2 years so wonderful. Above all, thanks to the Almighty God for showering his blessings on me.

TABLE OF CONTENTS

Page

1 INTRODUCTION 1

1.1 The Problem we Address 1

1.2 Existing Solutions 2

1.2.1 Unicast Solution 2

1.2.2 Network Layer Solution 2

1.2.3 Application Layer Solution 3

1.3 Our Strategy 5

2 OVERVIEW 8

2.1 System Goals 8

2.2 Thesis Contribution and Content Organization 9

3 THEORETICAL RATIONALE FOR TOPOLOGY CONSTRUCTION .. 11

3.1 Throughput Efficiency 11

3.2 Some Optimal Topologies 13

3.2.1 Fully Connected Topology 13

3.2.2 Chain Topology 14

3.3 Balanced Mesh 15

3.4 Cascaded Balanced Mesh 19

3.5 *b*-Unbalanced Mesh 21

3.5.1 Procedure to join a *b*-Unbalanced Mesh 22

3.5.2 Procedure to leave a *b*-Unbalanced Mesh 24

4 SYSTEM ARCHITECTURE 26

4.1 System Overview 26

4.1.1 Node Classification 26

TABLE OF CONTENTS (Continued)

Page

4.1.2 Component Interaction 27
4.2 Features of the System: 28
4.3 Advantages of the System 29
4.4 Drawbacks of the System 31

5 SYSTEMWIDE COMPONENTS 32
5.1 Wrapper Subsystem 32
5.2 Message Subsystem 34

6 SUPERNODE DESIGN 40
6.1 Mesh Manager 40
6.2 Supernode Session 43
6.3 SuperNode Design 47
6.4 Object Oriented Principles used in design 53

7 SUPERNODE WORKING 58
7.1 Handling list session requests: 58
7.2 Handling host session request: 58
7.3 Handling join session request: 59
7.4 Handling peer leave requests or peer fail messages: 61
7.5 Mesh Optimization 62

8 PERFORMANCE EVALUATION 65

TABLE OF CONTENTS (Continued)

Page

8.1 Small Scale Experiments 65
8.1.1 System Throughput Evaluation 65
8.1.2 Packet Delay Evaluation 67
8.1.3 Peer Join Evaluation 68
8.1.4 Peer Leave Evaluation 73
8.1.5 System Throughput Evaluation of an Optimized Mesh 75
8.1.6 Packet Loss Evaluation 78
8.2 Large Scale Simulation 79
8.2.1 Throughput Efficiency 79
8.2.2 Robustness Evaluation 79

9 RELATED WORK 83

10 FUTURE WORK 86

11 CONCLUSION 88

BIBLIOGRAPHY 89

APPENDICES 92
APPENDIX A First Appendix 93
APPENDIX B Second Appendix 101

LIST OF FIGURES

Figure Page

1.1 *Video delivery from a single source to multiple receivers* 1

1.2 *Replicated Unicasting* 3

1.3 *IP Multicast* 4

1.4 *Vanilla Multicast* 5

1.5 *Examples of (a) an overlay multicast tree; (b) a mesh topology* 6

3.1 *Chain topology with throughput efficiency of 0.55.* 12

3.2 *Fully Connected Topology* 14

3.3 *Illustration of balanced mesh construction.* 16

3.4 *A cascaded 2-balanced mesh.* 20

3.5 *(a) The mesh immediately before the destruction of the small mesh; (b) immediately after the destruction of the small mesh.* 22

4.1 *Component Interaction during a join request* 28

5.1 *Class Diagram for Message Subsystem* 35

6.1 *Class Diagram for Mesh Manager* 41

6.2 *(a) BMesh linear state for b = 4 (b) BMesh span state for b = 4* 42

6.3 *Communication between Session Manager and Mesh Manager* 44

6.4 *Class Diagram for SuperNode* 49

7.1 *Sequence of events for successful session hosting* 59

7.2 *Sequence of events at the SuperNode for a join request* 60

7.3 *Sequence of events at the SuperNode for a leave request* 62

7.4 *(a) 15 peer topology for b = 4 without optimization (b) 15 peer topology for b = 4 with optimization* 63

8.1 *(a) Average Down Speeds for a 15-Peer Topology (b) Average Up Speeds for a 15-Peer Topology (c) Comparison of file transfer time for a 15-Peer topology* 66

8.2 *Average Packet Delay for a 21-Peer topology* 68

Figure Page

8.3 *(a) Average join times at different points in the algorithm for 21-Peer topology (b) Number of peers joining at different logical positions for 21-Peer topology (c) Cumulative average join time for 21-Peer topology* 70

8.4 *Data flow at SuperNode for join requests for 21-Peer topology* 72

8.5 *(a) Average leave times for a 21-Peer topology (b) Data flow at SuperNode for leave requests for a 21-Peer topology* 74

8.6 *(a) Average Down Speeds for a 15-Peer Topology (b) Average Up Speeds for a 15-Peer Topology* 76

8.7 *(a) Total data flow for join for 15-Peer topology (b) Average join time for 15-Peer Topology* 77

8.8 *Packet Loss Evaluation* 78

8.9 *(a) Efficiency vs. variation; (b) Efficiency vs. out-degree for different data dissemination schemes.* 80

8.10 *(a) Percentage of affected nodes as a function of percentage of failed nodes for different values of branching factor b; (b) Percentage of affected nodes as a function of percentage of failed nodes for different allowable failures with* $b = 4$. 81

1. INTRODUCTION

1.1. The Problem we Address

In recent years there has been a surge in the number of applications that make use of streaming media. Video Conferencing, remote presentations and distance learning are stand out examples of such applications. Also a large number of systems that make use of peer resources have been developed. Commonly called Peer to Peer (P2P) networks, these systems make use of untapped peer potential to facilitate sharing of information and quick data delivery. These applications have largely influenced our lives and have the potential to affect us even more. Imagine you being able to make a presentation from your office to an audience world wide. These sort of applications have truly made the world a small place.

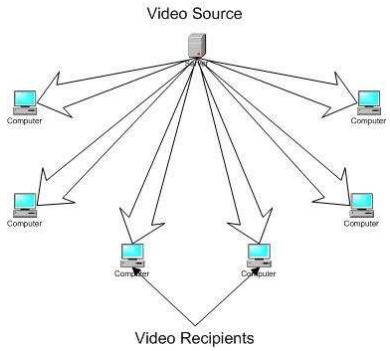


FIGURE 1.1. *Video delivery from a single source to multiple receivers*

To make the problem we are trying to address clearer, let us take the following example. Consider the problem shown in Figure 1.1. It involves video

data delivery from a single source to multiple destinations spread across the world. There have been multiple approaches to solve the problem. Let us go through them briefly.

1.2. Existing Solutions

In this section, we look at existing solutions to the single source multiple receiver problem mentioned earlier and see the drawbacks associated with each of them.

1.2.1. Unicast Solution

Most high-level network protocols (TCP or UDP) only provide unicast transmission service and therefore, nodes can send data to only one other node. All transmission with a unicast service is inherently point-to-point. The solution to the single source multiple receiver problem using unicast is as shown in Figure 1.2. In this case, the source creates N (where N is the number of receivers) copies of the data it wants to transmit and sends it individually to each destination node. This solution is not very efficient due to 2 main reasons: a) Duplicate packets b) Inability of the source to keep up with the high bit rates required by many applications such as video streaming.

1.2.2. Network Layer Solution

In case of replicated unicasting, the major drawback was the creation of duplicate packets. A network layer solution to this problem is called IP Multicast. In IP Multicast, the onus of data delivery to multiple destinations is moved to the

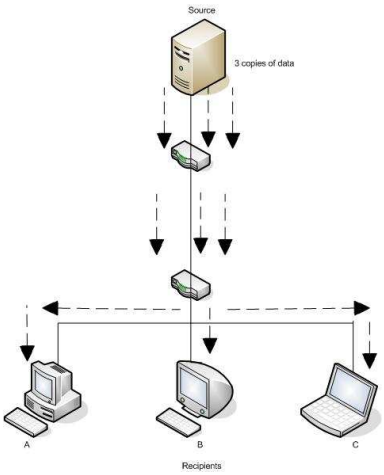


FIGURE 1.2. *Replicated Unicasting*

routers. The source generates a single packet and addresses it to the multicast address to which all destination nodes are subscribed. The routers handle details of how to route these multicast packets. The drawback of this system is that the job of data delivery is moved to the routers which are already stressed out performing basic functionality. Also the solution using IP multicast is not scalable due to compatibility issues across Autonomous Systems (AS). Finally the multicast tree topologies do not allow for optimum throughput and there was room for improvement. Figure 1.3 shows how a source can transmit data to multiple receivers using IP multicast.

1.2.3. Application Layer Solution

The application layer solution is also known as Overlay multicast. Overlay multicast aims at shifting the onus of data delivery and node organization from routers to end systems. The advantage of overlay multicast is that physical

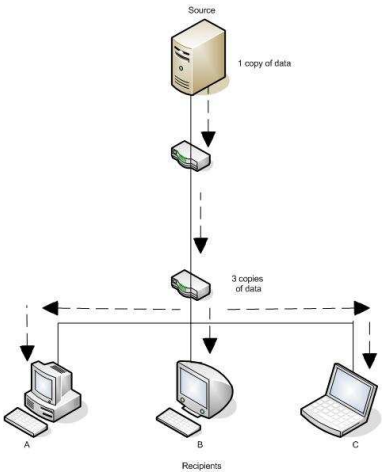


FIGURE 1.3. *IP Multicast*

routers need not support complex multicast operations. Instead, packet routing and forwarding are logically done at the application layer, which leads to easy deployment across different AS(es). However, overlay multicast techniques are sub-optimal since identical packets may travel on the same physical link due to the inability of the application layer to control the underlying routing. In addition, overlay multicast is not optimal in terms of throughput since the leaf nodes do not contribute their bandwidth to the system.

Most of the current streaming applications use overlay network models to disseminate information. One such system, the Vanilla Multicast is shown in Figure 1.4. Even in case of Vanilla Multicast, only the bandwidth of internal nodes is used while none of the leaf nodes contribute to the system upload capability. For most practical purposes, the fan out for a node in case of an overlay tree will be around 4. For fan out b less than 4, the tree is not robust and also the average delay for packets to reach the leaf node is large. However, as b increases, the number of nodes at the leaf level increases and hence there is more unused

bandwidth in the system. Thus, even some overlay multicast systems do not deliver optimal performance.

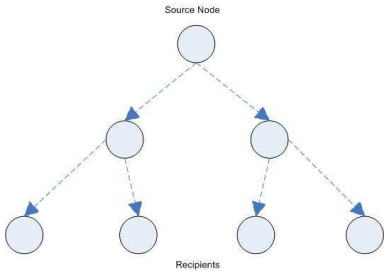


FIGURE 1.4. *Vanilla Multicast*

1.3. Our Strategy

As seen in Section 1.2.3, overlay multicast is not optimal in terms of throughput since the leaf nodes do not contribute their bandwidth to the system. For example, consider Figure 1.5(a) in which the source wants to disseminate a large file to all nodes in an overlay network. If the link bandwidth between the source and node B is 100 kbps, then all the destination nodes below B (B's children) will receive packets at the maximum rate of 100 kbps, even though the sending bandwidth of B can be much larger than 100 kbps, e.g. 10 Mbps.

Let us now consider a different topology in Figure 1.5(b) where there are additional links between the destination nodes, particularly, one link from C to B. Assume further that node C has bandwidth of 300 kbps to relay the traffic from the source to node B. If the set of packets that B receives directly from the source and the set of packets that B receives from C are completely disjoint, then B can forward the useful data to its children at the maximum rate of 400 kbps, a bandwidth improvement of four times over the overlay multicast tree

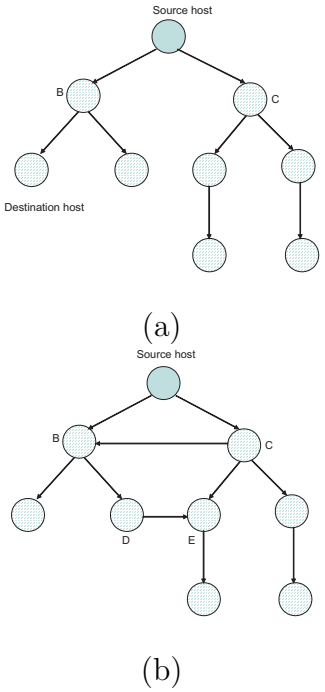


FIGURE 1.5. *Examples of (a) an overlay multicast tree; (b) a mesh topology*

approach. Therefore, using proper data partitioning techniques and topology, can help improve average throughput significantly.

Traditional data dissemination techniques place bandwidth constraints on individual links. This model does not reflect many networks such as DSL or wireless networks, where there is limited upload bandwidth but little restriction on the download bandwidth. The thesis focusses on the data dissemination problem in a source constraint network. In a source constraint network, bandwidth constraint is associated with a node's upload bandwidth, and hence for most practical purposes, a node's download bandwidth can be considered infinite. The unique feature of a source constraint network is that a node can allocate its sending rates to each of its neighbors as long as the total sending rate does not exceed the

capacity. A node can have multiple links to other nodes, but the total bandwidth is constrained to a maximum value. A lightly loaded P2P network [6] [7] of DSL subscribers is an example of a source constraint network since the bandwidth constraint is the node's upload physical bandwidth, e.g. 250 kbps which is much smaller than its download bandwidth. A peer may have multiple connections to other peers but its total upload bandwidth must be smaller than its physical bandwidth. Another example of source constraint network is a wireless network where the power constraint is placed on each node. Hence, a wireless node may be restricted to send data at a total maximum rate to other nodes, independent of the number of its neighbors.

After a careful study, we determined that overlay multicast depends upon a couple of key factors to deliver competent performance levels.

1. Topology

2. Contributions from each participating node towards increasing the system throughput.

We endeavor to address the above issues by presenting an algorithm to design an application layer mesh that exhibits near optimal throughput by having all nodes, including the leaves, contribute to the overall system throughput.

2. OVERVIEW

After studying the traditionally employed approaches towards disseminating video from a single source to multiple recipients and the drawbacks associated with each of these approaches, we propose a scalable topology capable of better throughput by having each participating node contribute towards the system throughput.

For the purpose of our discussion, we can safely make the following assumptions.

1. The download bandwidth of a node is larger than its upload bandwidth. Hence, the bandwidth bottleneck is due to the upload capacity of a node. This assumption holds true for the networks consisting of DSL subscribers or wireless networks.

2. We assume that upload capacity of all nodes is approximately the same in order to simplify our discussion. However we do provide ways to design a mesh in case of nodes with different upload capacities.

2.1. System Goals

The proposed topology and the associated data dissemination algorithm are designed to achieve the following:

1. Bandwidth is fairly distributed among nodes, i.e. the total receiving rate and sending rate of a node are equal.

2. A node can leave the network after it receives complete data without damaging the connectivity of remaining nodes.

3. End-to-end delay from the source to any node is small in order to support real-time applications.

4. Out-degree of any node is small for saving system resources.

5. Bandwidth usage of all nodes are optimal in the sense of average useful throughput, a quantity defined in Section 3.1.

2.2. Thesis Contribution and Content Organization

The thesis discusses the issues involved in topology construction and provides an algorithm for constructing an efficient data dissemination topology. In addition, the thesis provides details of the design of the SuperNode (a special node in charge of node and topology management) and looks at different components of the SuperNode in detail. The thesis also focusses on the working of the SuperNode and how requests from peers are handled by the SuperNode. Finally, a series of experiments evaluate the performance of the overall system and look at the mesh management overhead at the SuperNode in various scenarios.

The rest of the content is organized as follows. In Chapter 3, we define the notion of throughput efficiency to measure the performance of any source constraint network topology and data dissemination algorithm. In addition to this, we propose algorithms for constructing different topologies and associated data dissemination algorithms that maximize the throughput efficiency and at the same time, maintain a reasonable trade-off between delay and out-degree. The chapter on System Architecture gives a broad overview of the system designed, along with features, advantages and disadvantages of the system. This is followed

by Chapter 5 which discusses design of common components which are utilized by the entire system. Next, in Chapter 6 we discuss design issues involved while developing the SuperNode and provide overview of the current design. In Chapter 7 we discuss the working of the SuperNode and how the SuperNode performs desired functions in various scenarios. Following this, we present the results of some large scale simulations and small scale deployment across PlanetLab [25] nodes in Chapter 8. Finally, we discuss our conclusions, look at related work in the area of data dissemination and also state future work to improve the system.

3. THEORETICAL RATIONALE FOR TOPOLOGY CONSTRUCTION

The content presented in this chapter is an outcome of the joint work of Rohit Kamath [26] and myself. The discussion in this chapter is similar to that presented in [27].

3.1. Throughput Efficiency

To measure the performance of different data dissemination schemes in source constraint networks, we define the notion of throughput efficiency:

Definition 1: Throughput efficiency is defined as

$$E \triangleq \frac{\sum_{i=0}^{i=N} S_i}{min(\sum_{i=0}^{i=N} C_i, NC_0)} \tag{3.1}$$

where 0 denotes the source node, $i = 1...N$ *denote* N *destination nodes,* S_i *and* C_i *are the* useful *sending rate and the sending capacity of node* i*, respectively.*

The numerator is the sum of the *useful* sending rate of all the nodes in the network. The *useful* sending rate is the rate at which data is sent to neighbors such that this data is disjoint from other data that the neighbor receives. In a data dissemination scheme, if nodes receive duplicate data, then the scheme is not considered to be optimal. Hence a node i stops sending data to its neighbor once it discovers that its neighbor is already receiving that data from another node. In this case, the actual sending rate of a node would be equal to the *useful* sending rate. The denominator in Definition 1 is the minimum of 2 quantities. They are a) the total maximum sending capacity of all the nodes b) the maximum

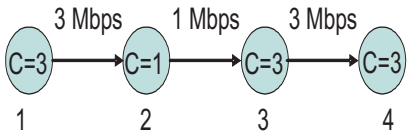


FIGURE 3.1. *Chain topology with throughput efficiency of 0.55.*

receiving capacity. Clearly efficiency is 1 when all nodes are sending at their maximum capacity. However, the sending capacity of all nodes might be larger than the allowable receiving rate, as this rate is dictated by the rate at which data is injected into the system. To ensure that the efficiency does not reduce in a situation where the network has a large capacity but small injected data rate, the denominator considers the minimum of the 2 values.

Let us consider an example to see how this notion of efficiency can be used to measure effectiveness of various topologies. Figure 3.1 shows 4 nodes connected to each other in a chain topology. To disseminate data, node 1 sends packets at 3 Mbps to node 2. The upload capacity of node 2 is only 1 Mbps and it forwards data at that rate to node 3. Since the receiving rate at node 3 is 1 Mbps, it cannot do better than that even though its upload capacity is 3 Mbps. It forwards data at 1Mbps to node 4. Thus throughput efficiency in this case is $\frac{3+1+1}{3*3} = 0.55$. In the above scenario, node 2 was acting as a bottleneck. If this node is moved to the end of the chain, the throughput efficiency of the system is $\frac{3+3+3}{3*3} = 1.0$. From the above example it is clear that the useful sending rate depends on the network topology.

Theorem 1: Throughput efficiency $E \leq 1$ *for any topology and data dissemination algorithm.*

Proof for this Proposition can be found in APPENDIX A.

In the following sections, we propose a number of topologies that maximize the throughput and at the same time, achieve trade-off in delay and out-degree.

3.2. Some Optimal Topologies

In this section we study two simple and intuitive optimal topologies and note their drawbacks.

3.2.1. Fully Connected Topology

A fully connected topology for one source and N destination nodes is one in which any node i is connected to all the other N nodes. Figure 3.2 shows a fully connected topology. In this case there is no restriction in the amount of resources consumed per node. To disseminate data, the source divides the data into N disjoint parts and sends each part down one of its out-going connection at a rate C/N. When the nodes receive data from the source, they forward it to the other $N-1$ destination nodes at a rate C/N. The drawback of this system is that it is not scalable, because the amount of system resources consumed increases linearly with N.

Theorem 2: For a fully connected topology, the following properties hold.
(a): Throughput efficiency of this scheme $E = 1$.
(b): The maximum node delay D is constant.
(c): Node insertion and deletion for this algorithm can affect at most N nodes where N is the number of destination nodes.
(d): The out-degree of each node is at most $N - 1$ where N is the number of destination nodes.

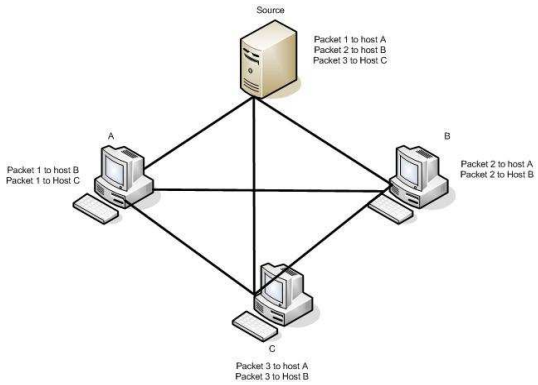


FIGURE 3.2. *Fully Connected Topology*

The properties can be proved as shown in APPENDIX A

Thus in this topology the time taken to receive a packet is greatly reduced but the out degree of nodes and the number of nodes affected by leave or join is high.

3.2.2. Chain Topology

A chain topology consists of a chain of all the N destination nodes. The source acts the head of this chain and forwards data to node 1. This node in turn forwards data to its neighbor and so on. A chain topology is shown in Figure 3.1. For a chain topology, the out-degree of each node is 1 and nodes forward data to their neighbors, at the rate of C bps.

Theorem 3: For a chain topology, the following properties hold.
(a): Throughput efficiency of this scheme $E = 1$.

(b): The maximum node delay D is $O(N)$ where N is the number of destination nodes.

(c): Node insertion and deletion for this algorithm affects constant number of nodes.

(d): The out-degree of each node is constant.

The properties can be proved as shown in APPENDIX A

Thus this topology is efficient in terms of number of outgoing connections and number of nodes affected by a join or leave but increases packet delay.

3.3. Balanced Mesh

The 2 optimal topologies presented in the previous section were extreme cases for minimizing node out-degree or delay. The topologies did not consider a trade off between out-degree and delay to achieve better overall performance. Our aim is to construct a topology that provides high throughput efficiency with low out-degree and small delay. We assume that upload bandwidth of all nodes is similar. The proposed topology takes form of a balanced mesh. A balanced mesh is a balanced tree in which the leaf nodes connect to each other and to their ancestors in a systematic manner to provide efficient data dissemination. Figure 3.3 shows an example of a balanced mesh with $b = 2$.

For the Figure 3.3, the data dissemination algorithm is as follows. The source partitions the data into two disjoint parts and sends it down the left and right subtree. Thus all nodes including the leaf nodes receive data that was sent down their subtree. Each leaf node then connects to a leaf node in the other group to receive data from that group. Hence the leaf nodes are the first to

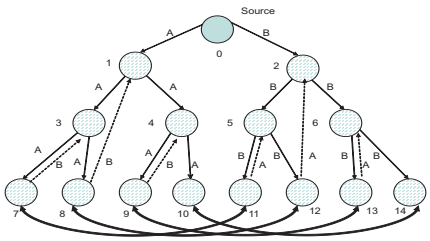


FIGURE 3.3. *Illustration of balanced mesh construction.*

receive complete data. Figure 3.3 shows nodes 7 and 11, 8 and 12, 9 and 13, 10 and 14 connected to each other. The internal nodes still do not have data from the other group. Note that the leaf nodes have made only 1 connection so far and hence they still have unused bandwidth. The leaf nodes forward the data from the other group to their ancestors and hence the internal nodes also receive complete data. A leaf node starts by trying to forward data to its parent. If its parent already receives data from the leaf nodes sibling, then the leaf node forwards data to its grandparent. For example, node 7 forwards data to its parent (node 3). Since node 3 already receives that data from node 7, node 8 forwards data to its grandparent (node 1). Node 9 forwards the data to its parent (node 4) while node 10 does not forward any data to its ancestor since there is no node in need to data in its group. The process is repeated by leaf nodes in the right subtree.

We now present the general algorithm for constructing a b-balanced mesh. The algorithm ensures that (a) all the nodes receive complete data, (b) no node has out-degree of more than b, (c) the number of hops from the source to the destination nodes is $O(\log_b N)$ where N is total number of nodes, and (d) throughput efficiency $E = 1$.

To describe the algorithm, we first label the nodes as shown in Figure 3.3. The nodes are labeled from low to high in a breadth-first manner. Within a level, the nodes are labeled from left to right.

Algorithm for constructing the balanced mesh is as follows.

1. Using the source as the root, first construct a balanced tree with each internal node having out-degree of b.

2. Label the leftmost group as group 0 and the rightmost group as group $b-1$. Assuming the tree has i levels, each leaf node j in the leftmost group is then connected to $b-1$ other leaf nodes in each of the different $b-1$ groups. In particular, node j in the leftmost group g is connected to nodes $k = j+b^{i-1}m$ where $m = 1, 2, ...(b-1) - g$. Also node k connects to j. This process continues for all groups g, ranging from 0 to $b-2$. These groups have groups to their right to whom they can connect.

3. Each leaf node (except the rightmost) of the same parent is connected back to its parent. The rightmost leaf node is then connected to its lowest ancestor without b incoming connections. Thus each parent of a leaf node will have b incoming connections. $b-1$ connections from its children and 1 connection from its parent. The grandparent will also have b incoming connections. 1 connection from its parent and $b-1$ connections from the rightmost child of its first $b-1$ children. This process continues till all ancestors have exactly b incoming connections. At the end of topology construction, the rightmost leaf node in each of the b groups does not connect back to its ancestors. Hence the system still has $b(C/b) = C$ bps unused capacity.

The pseudocode for construction of the Balanced mesh for N nodes is presented in APPENDIX B. Given the balanced mesh, the data dissemination algorithm is as follows.

1. The source splits the data into b partitions. It sends one partition down each of the b groups at the rate of C/b bps per group. The internal node forwards this data each of its children at the rate of C/b.

2. A leaf node forwards data got from its parent to $b-1$ leaf nodes of other groups to whom it is connected, at the rate of C/b. Now all leaf nodes have complete data.

3. A parent receives data from $b-1$ of its children. Each child forwards a different partition and hence the parent receives all the b partitions (since it already has the partition sent down its group). All the ancestor nodes also receive the complete data from the leaf nodes with each leaf node forwarding a different partition.

Theorem 4: For a balanced mesh, the following properties hold.

(a): Throughput efficiency of this scheme $E = 1$.

(b): The out-degree for each node is at most b.

(c): The maximum node delay D is $log_b((b-1)N+b)+1$ where N is the number of destination nodes.

The properties can be proved as shown in APPENDIX A

3.4. Cascaded Balanced Mesh

The balanced mesh provides a topology that results in optimal data dissemination. But for building a balanced mesh, the number of nodes must be of the form $(b^i-1)/(b-1)$ where $i, b \in 2, 3, ...$ This puts a restriction on the number of nodes that are needed to build an optimal topology. To overcome this problem, we introduce the concept of Cascaded Balanced Mesh which allows us to create an optimal topology with any arbitrary number of nodes. The idea of a Cascaded Balanced Mesh is to cascade a series of balanced meshes to accommodate any number of nodes.

As seen in Section 3.3, the rightmost leaf node in each group does not forward data to its ancestors. Hence each rightmost node in the b groups has C/b bps unused bandwidth. Thus there is a total unused bandwidth of C bps. Using this unused bandwidth, a new balanced mesh can be created. The root of the new mesh can be connected to the b rightmost leaf nodes of the previous mesh. Thus the cascaded mesh root receives data at the rate of C bps. The root can then disseminate data to all destinations below it, using the algorithm for a balanced mesh. Figure 3.4 shows an example of cascaded 2-balanced mesh consisting of 23 nodes. The rightmost nodes in each group, node 10 and node 14, of the first balanced mesh forward data to 15. Then a balanced mesh is constructed using 15 as the root. This mesh has nodes 19 and 21 with unused capacity. These nodes forward data to node 22 which is a balanced mesh with a single node.

The general algorithm for construction of a cascaded b-balanced mesh consisting of N destination nodes is as follows.

1. Construct a b-balanced mesh with the depth $i = \lfloor log((b-1)N + b) \rfloor - 1$. This step constructs the deepest b-balanced mesh without exceeding the

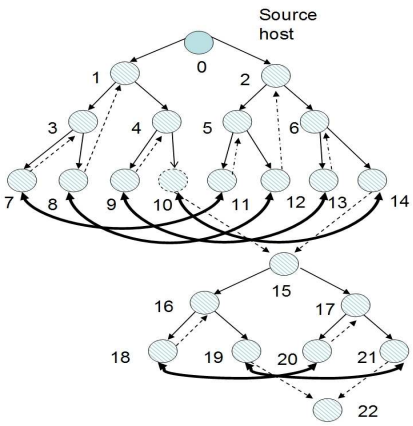


FIGURE 3.4. *A cascaded 2-balanced mesh.*

number of nodes. If there exists a previous b-balanced mesh, connect the b rightmost leaf nodes with extra bandwidth to the root of a newly created balanced mesh.

2. Set $N = N - (b^{i+1} - 1)/(b - 1)$. This is the number of remaining nodes.

3. If $N = 0$, stop. Otherwise, go back to step 1.

The pseudocode for construction of the cascaded balanced mesh for N nodes is presented in APPENDIX B. Since the construction of the cascaded balanced mesh is based on that of a balanced mesh, the properties of the cascaded balanced mesh are similar.

Theorem 5: For a cascaded balanced mesh, the following properties hold.

(a): Throughput efficiency $E = 1$.

(b): The delay is $O((log_b N)^2)$.

(c): The out-degree for each node is at most b.

The properties are proved in APPENDIX A

3.5. b-Unbalanced Mesh

There are two drawbacks with the cascaded balanced mesh. First, the delay is rather large, i.e. order of $(\log_b N)^2$. Second, if nodes enter and leave incrementally, a large portion of mesh may have to be rebuilt. In this section, we introduce a new construction that reduces the delay and enables nodes to join and leave incrementally with minimum effect on the mesh.

Similar to Section 3.4, the new algorithm uses the cascaded balanced mesh to accommodate the new nodes. For convenience, we denote the mesh containing the source node as *primary* mesh and other meshes connected to the primary mesh as *secondary* meshes. The idea of achieving low delay is to keep only a small number of secondary meshes by limiting the total number of nodes in the secondary meshes to a just $b^2 - 1$. This node limitation allows quick reconstruction of the secondary meshes to accommodate nodes entering and leaving the topology. When the number of nodes in the secondary meshes equals to b^2, they are destroyed and their nodes are then attached to the primary mesh at appropriate places to achieve high throughput efficiency and low delay. The secondary meshes is destroyed only when the number of nodes reaches b^2 because this is the smallest number of nodes that can be attached at right places in the primary mesh to maintain the throughput efficiency of 1.

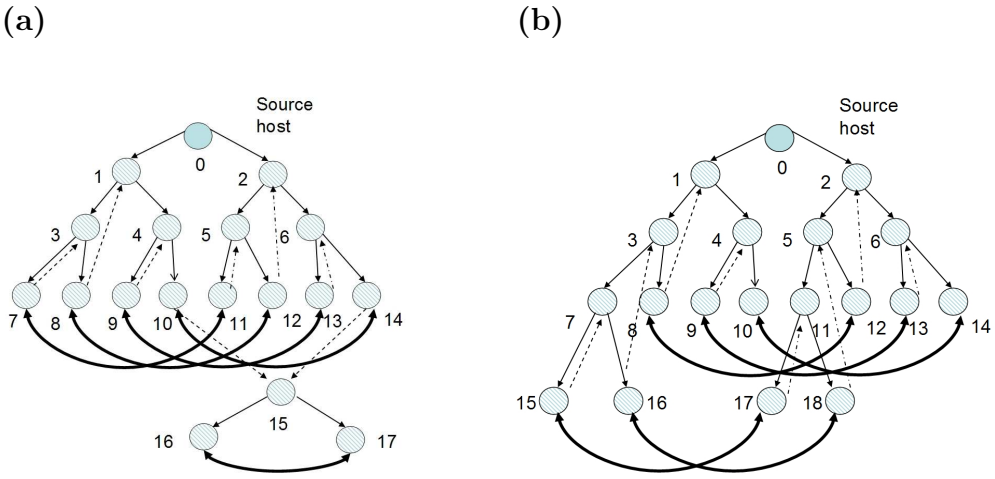


FIGURE 3.5. *(a) The mesh immediately before the destruction of the small mesh; (b) immediately after the destruction of the small mesh.*

3.5.1. Procedure to join a b-Unbalanced Mesh

Assume that we already have in place a balanced *primary mesh*. When a new node comes in to join the network, it acts as the root node of the *secondary mesh*. Now, when another node joins in, it is added to the secondary mesh using the algorithm for constructing the cascaded balanced mesh as described in Section 3.4. This process repeats as long as the number of nodes in the secondary meshes is lesser than b^2. Once the node count reaches b^2, we destroy the secondary mesh(es) and join the nodes to the primary mesh as described below.

Let there be j levels in the primary mesh before the secondary mesh is attached. There are 2 ways to attach the broken secondary mesh to the primary mesh.

1. If all leaf nodes of the primary mesh are at the same level, then attach the first b nodes to the leftmost node in the first group. The second b nodes will be attached to the leftmost node in the second group and so on.

2. If the leaf nodes of the primary mesh are at different levels (due to attachment of secondary mesh nodes earlier), then attach the first b nodes to the leftmost leaf node at a lesser depth, in the first group. Attach the other set of b nodes to similar positions in the remaining $b-1$ groups.

When b new nodes attach to a node P in the primary mesh, node P disconnects $b-1$ connections to other nodes in other $b-1$ groups and one connection that is used to forward data from another group to its ancestor. With the availability of b connections, node P can forward data from its parents to the b new nodes. The b new nodes are then connected to other nodes in other groups in a similar manner as described in the balanced mesh. Since the ancestor of node P no longer receives the data from the other branch, the rightmost node of the new b children of P will forward the data to P's ancestor. Now if more nodes join in, a secondary mesh is constructed again. After b^{j+1} new nodes joins, where j is the depth in the primary mesh, the primary mesh is balanced and its depth increases by 1. Figure 3.5 illustrates the incremental construction of a 2-unbalanced mesh. Initially, the primary mesh consists of 15 nodes. Figure 3.5(a) shows the resulted topology after 3 new nodes join. When the fourth node joins, the secondary mesh is destroyed and its nodes are attached to the nodes in the primary mesh. Nodes 15 and 16 are attached to node 7, nodes 17 and 18 to node 11. Nodes 7 and 11 are disconnected from each other. They also no longer forward data to their ancestors. Instead, they use these two extra connections to forward data to the new nodes. Nodes 15 and 16 then exchange the data with nodes 17 and 18 as

before. Node 15 also forwards data from another group to its parent (node 7) and node 16 forwards data from another group to its grandparent (node 3). Similar connections are made in the right subtree. Thus all the nodes receive complete data. The pseudocode for construction of the b-unbalanced mesh for incoming node i is presented in APPENDIX B.

3.5.2. Procedure to leave a b-Unbalanced Mesh

If the departing node belongs to the primary mesh, perform one of following steps

1. If there exists a secondary mesh, pick a node from the secondary mesh to replace the departed node. This step maintains the same structure for the primary mesh. Next, rebuild the secondary mesh(es).

2. If there is no secondary mesh and the departed node is not of the largest depth, pick a leaf node in the primary mesh with the largest depth to replace the departed node. Next, construct a secondary mesh consisting of $b^2 - 1$ nodes. These $b^2 - 1$ nodes are the siblings of the chosen replacement node, the nodes in other groups that connect directly to the chosen node, and their siblings. If the departed node is of the largest depth, node swapping is not necessary and a secondary mesh consisting $b^2 - 1$ nodes associated with the departed node is constructed.

3. If the departing node belongs to a secondary mesh, rebuild the secondary mesh. It can be proven that in the P2P mesh, the number of nodes affected by a node removal or insertion is at most O(b^2) and the delay is of O($logb$).

Thus, this topology is scalable as the management overhead for node joining and leaving does not depend on the number of nodes but on the branching factor b.

The pseudocode for tree reconstruction for a b-unbalanced mesh when node i leaves is presented in APPENDIX B.

Theorem 6: For a b-Unbalanced mesh with N destination nodes, the following properties hold.

(a): Throughput efficiency $E = 1$.

(b): The delay D is at most $\lfloor \log_b(N+1) \rfloor + 3b - 4$.

(c): Node insertion and deletion for this algorithm can affect at most $b^2 + 2b$ nodes.

(d): The out-degree for each node is at most b.

These properties are proved in APPENDIX A.

4. SYSTEM ARCHITECTURE

The architecture presented in this chapter is similar to the one presented in [28].

Although the proposed b-unbalanced mesh can be built in a completely distributed way, we believe a hybrid P2P architecture offers many more benefits. A hybrid P2P architecture enables scalable data dissemination among nodes and at the same time provides other benefits such as security, flexibility due to the centralized management, etc. We now briefly discuss our hybrid architecture.

4.1. System Overview

Our system follows an Object Oriented approach and the design is based on the responsibilities of entities.

4.1.1. Node Classification

Depending upon the responsibility that a node is going to take on in the proposed hybrid P2P network, a node is classified as either a SuperNode or a Peer.

1. SuperNode: A SuperNode is the controller of the system. It is a special node which handles all requests from other nodes for joining or leaving a session. It is the node in charge of providing all control information (neighbor list, session list, etc) to the peers. Its task is to maintain an accurate global view of the topology of a session. A SuperNode can handle multiple sessions, i.e. keep track of multiple topologies. Each session in progress is assigned a separate Session Manager which handles all requests for that session. The Session

Manager contains a Mesh Manager (Algorithm Component (AC)) which runs the algorithm to maintain the topology. When a node joins or leaves a session, the AC is invoked and it produces a list of affected nodes. (nodes which have to make changes to their connections due to the join or leave). The SuperNode uses this list returned by the AC to send out messages to affected nodes instructing them to take appropriate action.

2. Peer: A Peer is a node that is participating in a session hosted by the SuperNode. A peer is part of the topology for that session. The peer gets information about its neighbors from the SuperNode when it joins a session. Also, when a new node is added to the network, the peer's neighbors may get updated. In this case, the peer is informed about the new neighbors by the SuperNode through standard messages (The different messages will be discusses in Chapter 5).

In addition to these main roles, a node can be an entity which wants to get information about all the ongoing sessions on a SuperNodes. It is not a part of any session and hence not part of the P2P network, until it decides to join or host a session, after which it becomes a peer.

4.1.2. Component Interaction

The Figure 4.1 shows the various components of the system and their interaction during a join session request from the peer.

The main components of the system are: 1) SuperNode and 2) Peer. Within the SuperNode, there are multiple session managers responsible for each individual session. Each session has an AC associated with it. The figure also shows a session in progress and its logical image in the algorithm component. Let

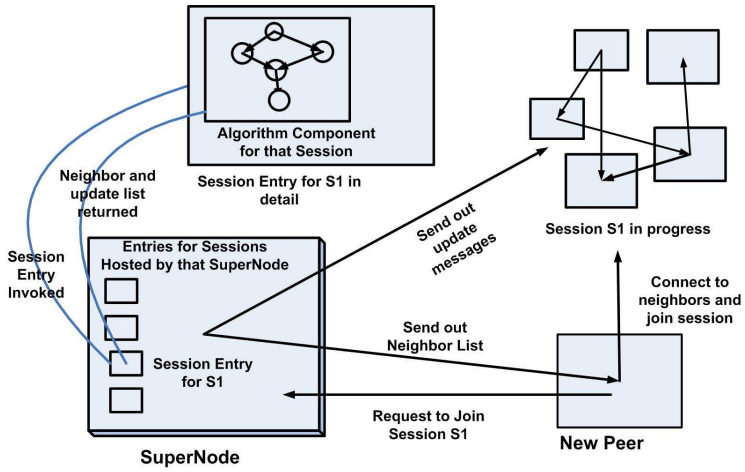


FIGURE 4.1. *Component Interaction during a join request*

us briefly go through how these components interact when a new node wants to join the session.

Since the SuperNode is the controlling authority of the system, the peer contacts the SuperNode when it wants to join the session. The details regarding all SuperNodes that are managing sessions is available in a published list of SuperNodes. The peer gets the SuperNode details this published list. Once the SuperNode receives a join request, it will determine which session the peer wants to join and invoke the corresponding session manager. The session manager will in turn call the AC which will add the node to the logical view of the mesh that it maintains. The SuperNode will then send updates to all the affected nodes and also provide the new peer with a set of neighbors.

4.2. Features of the System:

The features of the system are as listed below.

1. Support for multiple sessions: As part of the current design, the SuperNode can support multiple sessions. This is achieved by having a separate Session Manager in charge of each ongoing session. However there is an upper bound on the number of sessions that a SuperNode can support. Even the peer is capable of participating in multiple sessions.

2. Handling node failure (Heartbeats): The peer implements this feature wherein each peer monitors its incoming links to detect if any of its sources have failed. The peer gets the list of sources from the SuperNode and it has a thread dedicated to check these input links. If the peer detects a failed peer (indicated by absence of ping data for some stipulated value of time) then it informs the SuperNode about it and the SuperNode removes the failed peer from the mesh.

3. Mesh Optimization: In Chapter 2 we assumed that the upload capacity of all nodes is similar. However, this does not hold true in the real world scenario where nodes with different capacities may want to part of the same session. In this case, the SuperNode optimizes the mesh by placing all the high capacity peers close to the root of the mesh followed by the lower capacity peers. Currently this is done statically (based on capacity provided by the peers at join time) but it can be done dynamically (based on network conditions) as explained in Chapter 10.

4.3. Advantages of the System

There are a number of advantages of having a hybrid P2P system design, namely

1. Load Distribution: The design separates the work of topology management and data dissemination. The responsibility of the peer is now limited to forwarding data as instructed by the SuperNode. Thus the peer becomes very simple. The complexity of running the algorithm and any issues related to optimizing the performance of session are handled by the SuperNode.

2. Easy Control Information access: The SuperNode acts as the centralized point of access for all Control Information (list of sessions, neighbors when a node joins, etc). In pure P2P networks, in order to find session information, the peer has to flood the system with the request for a file and then each individual peer checks if it has that file available. By having a SuperNode, a peer contacts this SuperNode and gets information about all the files being streamed. Also the procedures for node join and leave is far more simplified by having a controlling entity like the SuperNode.

3. Security: Centralized control over nodes joining the system prevents malicious users from entering the network. The SuperNode can authenticate the nodes participating in a session. As an example, consider a confidential live meeting being streamed over the corporate network where only nodes with certain IP addresses are allowed to receive the video. In this case, the SuperNode handling this session can authenticate whether the node wanting to join the session has a valid IP address. In addition to this, the SuperNode has control over the types of files allowed to be streamed. Also only the SuperNode knows the algorithm being used to construct the mesh rather than having to divulge this information to any peer joining the session.

4. Flexibility: Hybrid architecture offers high flexibility in terms of management and upgradability. Suppose, we wish to upgrade the algorithm or

change the network topology or add more security/controlling features, we just need to make changes in the SuperNode. The rest of the system can remain the same. This is because (a) the SuperNode sends to the peers the standard messages and (b) the peers simply follows the instructions contained in the standard messages. In addition, the system has the advantage that any change need not be known to the peers because all the control is with the SuperNode. Thus the whole system is easy to manage and upgrade.

4.4. Drawbacks of the System

1. Single Point of failure: The SuperNode may seem to be the single point of failure for the system. Because all control decisions go through the SuperNode, if it fails then the whole system comes to a standstill. However, by ensuring that we have multiple SuperNodes and session information is smartly replicated, we can overcome this bottleneck.

2. Time overhead of contacting SuperNode: In pure P2P networks, the peers have greater control over whom to connect to or not and how to optimize their individual throughput. Also any changes to adjust to network conditions like congestion, link failure etc can be done by the peers themselves. In case of a hybrid architecture, since all decision making is done by the SuperNode, the peers will have to contact the SuperNode and wait for the SuperNode to make the changes. This introduces an additional overhead everytime a control decision has to be made.

5. SYSTEMWIDE COMPONENTS

There are components of the system that serve as helpers to the two main roles in the system, namely SuperNode and Peer. They are used by both SuperNode and Peer to fulfill their responsibilities. In this chapter we go through these components and study them in some detail. We stress on the utility of these components from the SuperNode's point of view but also keep in perspective their overall utility.

5.1. Wrapper Subsystem

The Wrapper Subsystem provides easy to use interfaces to network calls. These are wrappers written over the basic socket library routines to hide their complexity and provide a simple and uniform interface. The basic class in this system is of *Socket*, which represents an end point of communication. From this class we branch out two hierarchies, namely TCPSocket (which uses TCP protocol at transport layer) and UDPSocket (which uses UDP protocol at transport layer) . The TCPSockets are used for transfer of control information while the UDPSockets are used for transfer of data. Since the SuperNode is the controller of the system and mainly issues instructions (control information) to peers, it always uses TCPSockets. The peer, receives control information and also transfers data and hence makes use of both TCPSocket and UDPSocket. We will look into the details of TCPSocket in remaining part of this section.

1. TCPSocket: The *TCPSocket* class is derived from the *Socket* class. It inherits the primitive SOCKET descriptor (provided by the Operating System)

from the *Socket* class. It implements the send and receive functions of the *Socket* class.

2. TCPClientSocket: The *TCPClientSocket* class is derived from the *TCPSocket* class. In addition to being a *TCPSocket*, the ClientSocket provides the ability to connect to another socket. In case of a client socket, the port at which it is bound is chosen by the system at the time it is created. The SuperNode creates client sockets when it has to send control information to peers.

3. TCPServerSocket: The *TCPServerSocket* class is also derived from the *TCPSocket* class. The server socket listens to a port to accept incoming connections. At the SuperNode, the serversocket listens to the port that is advertised to all peers. All peers connect to this port to get the required service (hosting session, joining session, leaving session, etc). Being a serversocket, the TCPServerSocket has the ability to listen to a port and accept incoming connections. When a new connection is accepted by the TCPServerSocket, it spawns a new TCPClientSocket for further communication over that connection.

In addition to the classes mentioned above, there is the *MonitorSocket* class. *MonitorSocket* class provides the set of functions which allow objects of *Socket* class to be monitored and also help detect the presence of data in any one of these sockets. This class is used by the SuperNode to monitor the *TCPServerSocket* it creates and also monitor the client sockets that are spawned in response to requests from peers.

5.2. Message Subsystem

The Message subsystem consists of the messages which are used by the SuperNode and the peer to communicate with each other. Hence the knowledge of this component to both, the SuperNode and peer, is vital. Both the peer and the SuperNode expect messages in a format that has been agreed upon and use functions that are part of this subsystem to construct and parse these messages.

The figure 5.1 shows the class hierarchy for the message subsystem. The *Message* class is the base class for all the other message classes. The *Message* class specifies the receiver (*nodeId*) and the type of message (*msgType*). The type of a message could be one of *ackMessage, nodeFailMessage, etc.* The main idea behind having a separate hierarchy for messages is to isolate message creation and parsing from the SuperNode and peer. Each of these message objects knows how to create and parse itself. So the SuperNode or the peer need not know the actual format of any of the messages. Also any changes in the future to message formats will only entail changes in this subsystem. The SuperNode (or peer) identifies the type of the message (using the *msgType* field of the *Message* object) and invokes the parse function of that particular message object. We will briefly go through the different messages that are used by the system specifying details about their formats and the information they contain.

1. ACKMessage: This is a general acknowledgement (ACK) message sent by the SuperNode. It contains the *MessageID* and *sessionID*. An ACK is used in the following scenarios.

   (a) An ACK is sent in response to a host session request from the peer. In this case, an ACK indicates successful hosting of the session and the *sesionID* provides the id of the newly hosted session.

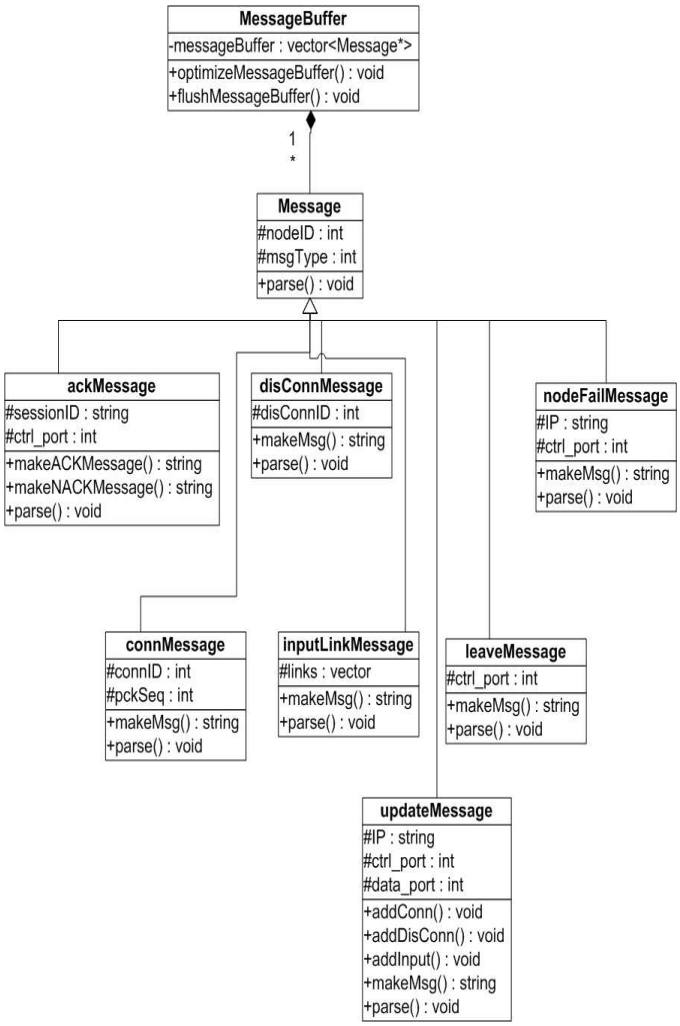


FIGURE 5.1. *Class Diagram for Message Subsystem*

(b) When a change is made to the mesh, certain nodes will have to update their connections. An ACK is also sent to tell the updated peers to finalize their new links. This mechanism will be further explained in the next chapter.

This class is used by the SuperNode to also make NACK messages. A NACK message is sent, when the SuperNode cannot process the request from a peer. A NACK is sent when a node requests the SuperNode to host a session but the SuperNode cannot handle anymore sessions or when a peer wants to join a session that is not hosted by the SuperNode. In addition to the *MessageID* and *sessionID*, the NACK message also contains an *error code* which indicates the cause of the NACK.

2. connMessage: The *connMessage* is a connection message. It is generated by the Mesh Manager and placed in the Message Buffer (refer point 8). The *connMessage* indicates the id of the node to connect to, along with the packet sequence number of data packets to forward to that node. The objects of this class are used to create the *updateMessage* (refer point 5).

3. disConnMessage: The *disConnMessage* is a message instructing a peer to disconnect from another peer. It is also placed in the Message Buffer by the Mesh Manager. The *disConnMessage* contains the id of the peer with which connection is to be ended. The *disConnMessage* is also used to create the *updateMessage*.

4. inputLinkMessage: The peers not only keep track of peers that they are supposed to send data to but also keep note of peers from whom they are supposed to receive data. The *inputLinkMessage* for a peer, contains a list

of sources for that peer. This list is in terms of peer ids. The peer expects ping data from its sources and if it does not receive a ping for a set interval of time, it sends a *nodeFailMessage* (refer point 7). The sources to a peer are also sent as part of the *updateMessage*.

5. updateMessage: This message is sent by the SuperNode to the peers to inform them of any changes in their connections. When a peer joins a session, it expects an *updateMessage* from the SuperNode to indicate successful join. Also if a change to the mesh (due to node join, peer leave, etc) has resulted in a new or modified neighbor list for a peer, then it receives an *updateMessage*. The *updateMessage* gives information about the connections to be made or broken and also has information about new sources for a peer. The format for the *updateMessage* is as shown in the table below:

Message	Details
Message_ID	identifies it as update message
SessionID	indicates the session to which the message refers
C IP:ctrl_port:data_port tagID	indicates a new **c**onnection to be made to that IP
D IP:ctrl_port	instructs peer to **d**isconnect from another peer
I a IP:ctrl_port	indicates a new **i**nput source to be **a**dded
I r IP:ctrl_port	indicates **i**nput source to be **r**emoved from its source list
C IP:ctrl_port:data_port tagID	**c**onnection to be made to another peer

Let us study the *updateMessage* in some detail. In the message, the term *ctrl_port* refers to the control port of a peer. The peer receives control information like neighbors list, update list, ping data from sources, etc at this port. *data_port*, is the port used only for data forwarding. The *connection*

part of the message specifies both the data and control port of the peer with whom connection is to be established. This is because the peers exchange both control information and data with their neighbors. The *tagID* field, in the *connection* part indicates which packet to forward to the connection. In the rest of the message, a peer is specified using its IP and *ctrl_port*. The IP and control port provide a unique identifier to the peer. The ordering of instructions within the *updateMessage* is the same as that generated by the *MeshManger* so that the final result is a consistent topology. To construct the *updateMessage*, the *connMessage, disConnMessage and inputLinksMessage* objects are used. Recall that these objects have the information in terms of node ids. The Session Manager is responsible to convert these ids to network addresses and ports. We will study how this is done in the next chapter.

6. leaveMessage: The *leaveMessage* is sent by the peer when it intends to leave the session. It contains the sessionId of the session it wants to leave and the control port of the peer. This port is then used by the SuperNode to send the *leaveSessionACK*.

7. nodeFailMessage: The *nodeFailMessage* is used by a peer to intimate the SuperNode about a failed peer. The peer constantly monitors its sources to see if they are alive. If the peer fails to receive ping data from a source for a stipulated interval of time, then it reports the source as failed. The *nodeFailMessage* contains the IP address and the control port of the failed peer. This serves as the unique identifier which helps the SuperNode remove the failed peer from the session.

8. Message Buffer: The Message Buffer is a collection of *Message* objects. It is populated by the Mesh Manager when it is updating the mesh in response to a join or leave request. When there is a node addition or deletion, the Mesh Manager generates a set of *connMessages, disConnMessages and inputLinkMessages*. These messages are stored in the Message Buffer. The Session Manager then, parses the Message Buffer and creates an *updateMessage* for each affected peer. The *optimizeBuffer* function of the Message Buffer sorts the buffer by *nodeID*. The optimized Message Buffer is used by the Session Manager to send out messages to affected peers in a more efficient manner. The advantages of buffer optimization are discussed in the next chapter.

6. SUPERNODE DESIGN

The SuperNode is the controller of the system. It is a complex component which makes use of the functionality of the other classes to do its job. In this chapter we provide the details of the SuperNode design and discuss the different classes that compose the SuperNode.

6.1. Mesh Manager

The Mesh Manager is also called the Algorithm Component since its sole responsibility is to run the algorithm. It has no networking functions and is oblivious to any network level details like IP, ports, etc. Each Supernode session has its own Mesh Manager. The Mesh Manager retains the logical view of the session topology. The Mesh Manager itself is composed of different classes that work together to run the algorithm. Figure 6.1 shows the class diagram for the Mesh Manager.

The classes that make up the Mesh Manager are as follows:

1. Node: This class represents a single node in the network. It keeps track of its parent, children, links and references. References refer to the sources from which a node receives data. Links keep track of the cross links as well as the links back to the parent. This class also provides methods to update the children, parent, links or references of a node.

2. BMesh: The secondary mesh (cascaded mesh) can contain multiple height-1 balanced meshes. A *BMesh* represents a height-1 balanced mesh. A *BMesh* contains at most $b+1$ nodes at a time. The *BMesh* can be either in a linear state (when there are b or less nodes) or in the span state (when there are

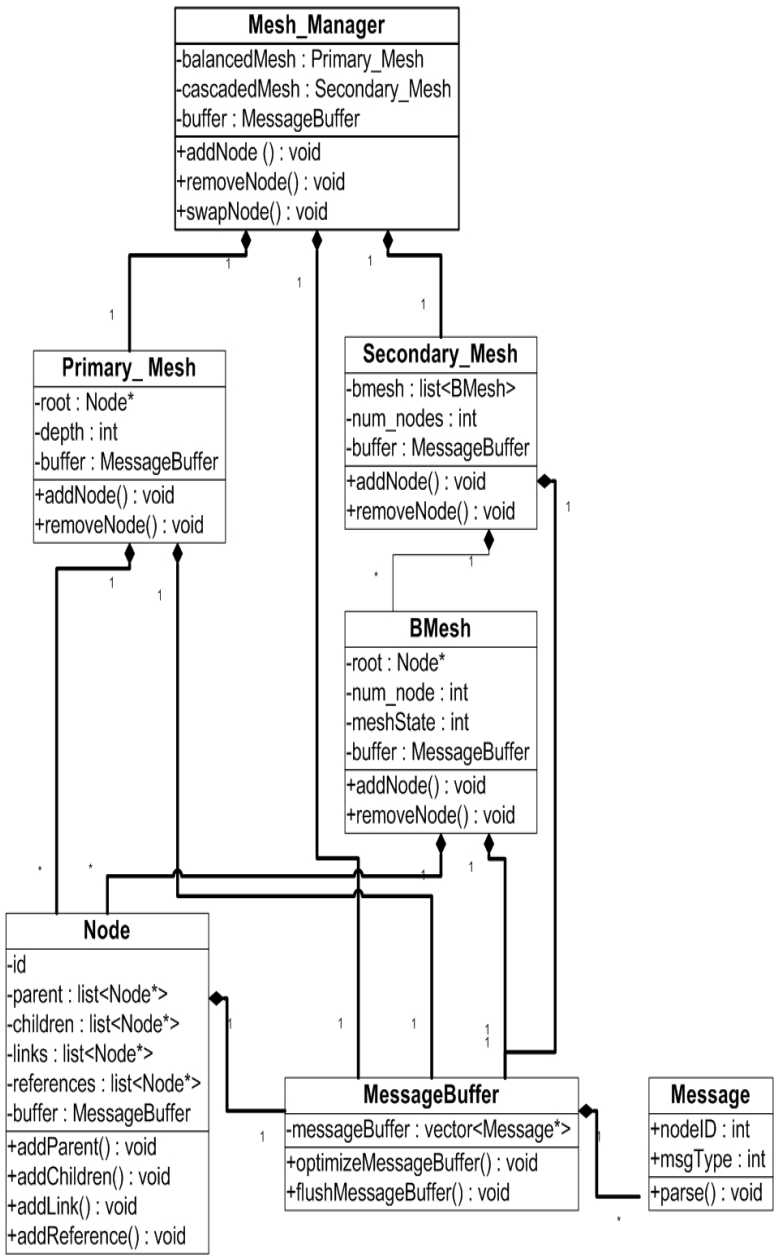


FIGURE 6.1. *Class Diagram for Mesh Manager*

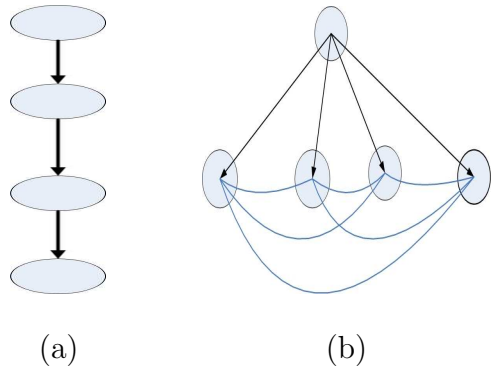


FIGURE 6.2. *(a) BMesh linear state for b = 4 (b) BMesh span state for b = 4*

$b + 1$ nodes). Figure 6.2 shows the linear and span state of a *BMesh* for $b = 4$. A secondary mesh can have at most $b - 1$ such height-1 balanced meshes. This class provides methods to add and remove nodes to the given *BMesh* while maintaining the desired topology (linear or span).

3. Secondary_Mesh: A *Secondary_Mesh* is composed of multiple objects of the *BMesh* class. The *Secondary_Mesh* class provides methods to insert or remove nodes from the secondary mesh. Node insertion or removal mainly involves identifying the *BMesh* to which the node belongs (or will belong) and then perform an insert or remove on that *BMesh*.

4. Primary_Mesh: The *Primary_Mesh* is responsible for maintaining the balanced mesh. Every time the secondary mesh breaks, the resulting b^2 nodes are attached to the primary mesh. The class provides methods to systematically arrange these b^2 nodes. This class also provides methods to remove a node from the balanced mesh. This may require swapping with another node in the secondary mesh.

5. Mesh_Manager: The *Mesh_Manager* class represents the entire algorithm component to the external world. It is composed of the *Primary_Mesh* and *Secondary_Mesh*. It provides methods to add and remove nodes from the mesh. This class also provides methods to optimize the mesh to improve system throughput. We will discuss the mesh optimization in the next chapter.

6.2. Supernode Session

Each session hosted by the SuperNode is handled by a separate Supernode session. It is also called the Session Manager since it completely manages one session and handles all the requests pertaining to that session. As shown in Figure 6.4, the Supernode session is inherited from the Session class. It inherits the attributes that provide details of a session. This base class is shared by both the peer and SuperNode to maintain session information like sessionId, filename, session protocol, etc. These details are returned to the peers seeking information about sessions handled by a SuperNode. In addition to this, a Supernode session is made up of the following components:

1. Mesh Manager: Every Supernode session has one instance of the Mesh Manager running. The Mesh Manager structure is as described in the previous section. It is responsible for maintaining the logical view of the topology. In order to add a node, the Supernode session calls the *addNode* function of the Mesh Manager which takes in the id and capacity (whether it is DialUp, T1, DSL, etc) of the node as parameters. The capacity of a node is used while performing optimization of the mesh. Similarly, to remove a node, the Supernode session calls the *removeNode* function of the Mesh Manager with

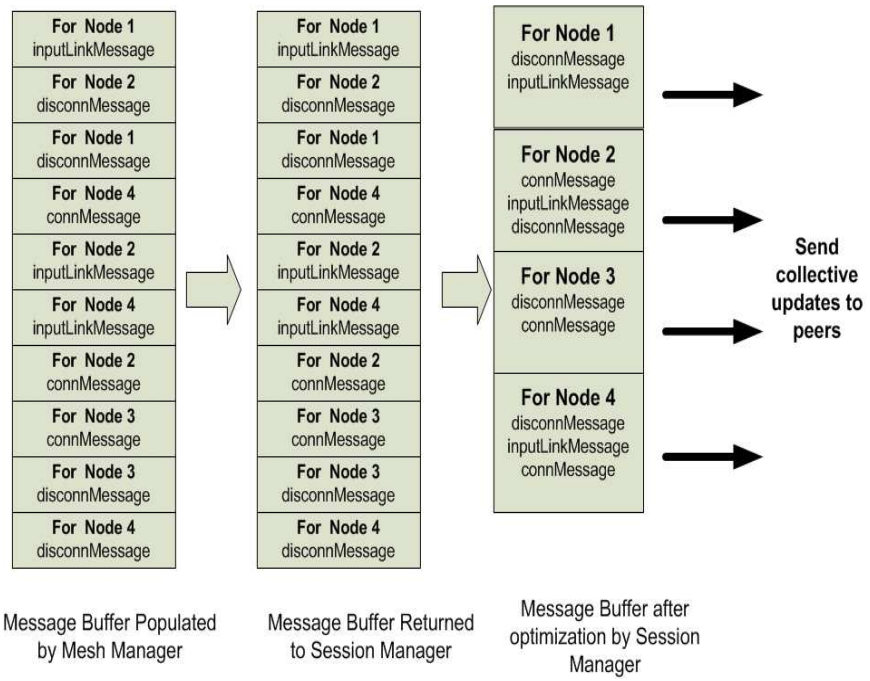


FIGURE 6.3. *Communication between Session Manager and Mesh Manager*

the id of the node to be removed.

Communication between Supernode session and Mesh Manager:

When the Mesh Manager gets a request from the session manager, it performs the desired operation (like *addNode, removeNode or swapNode*). The procedures to add, remove or swap nodes involves changing the mesh. When the mesh is changed by the Mesh Manager, messages for peers are generated. The Mesh Manager stores these messages in the Message Buffer. These messages are in the form of *Message* class objects. Specifically, the different messages put into the Message Buffer are *ConnMessage, DisConnMessage and InputLinkMessage*. Next, the control returns back to the Session Manager. Now, the Session Manager accesses this Message Buffer to send updates.

Before sending updates, the Session Manager optimizes the Message Buffer. Optimization involves iterating through the Message Buffer and grouping messages intended for the same peer. The final stage in Figure 6.3 shows the optimized Message Buffer. The destination for a particular message is stored in the *nodeID* field of the *Message* object. The optimization is a customized sort on the Message Buffer based on *nodeID* . The Mesh Manager is designed in such a way that the order in which messages are put into the buffer is important and needs to be preserved. Hence ordering of messages for a peer should be preserved after sorting. So, a stable sort is required. The last 2 stages in Figure 6.3 show how the Message Buffer is optimized. Note that order of messages for each node is preserved from stage 2 to stage 3 of Figure 6.3. This optimization is important because it allows the session manager to open a socket and send all messages for a peer at once rather than opening sockets everytime there is a message for that peer. The optimization is an important gain as control messages are sent using TCP and opening and closing TCP sockets is expensive as it involves the TCP three way handshake for establishment of connection. Once the Message Buffer is arranged in the manner described above, the session manager starts sending *update* messages to destination nodes. Following this, the buffer is flushed for use by the Mesh Manger for the next request. Figure 6.3 shows the structure of the Message Buffer as populated by the Mesh Manager. The Message Buffer shown can be considered to be a stack of *Message* objects. As seen there is no fixed ordering of messages at this stage. This buffer is returned to the Session Manager. Finally, the Session Manager combines messages intended for the same peer, while maintaining the relative order of messages, before sending out updates.

2. IPMappings: A node's complete information is specified in terms of its IP address, control port and data port. This information is stored in a structure, *node_info*. When a new node joins the session, it gets assigned a node id. The Mesh Manager uses the node ids to perform all operations. Thus, a mapping of ids to *node_info* is required. *IPMappings* stores the mapping of the ids to *node_info*. Each session object maintains a map of all the peers in the session that it is handling. It is a crucial component as it performs the translation of ids to actual network destinations. As soon as a join request is accepted by the session manager, an entry is created in the mapping. The objects of *Message* class present in the Message Buffer, contain all information in terms of node ids. The session manager extracts node ids from these objects, looks up the mapping and determines the corresponding IP addresses and port numbers. Next it sends the message to the destination with all ids converted to IP addresses and port numbers.

3. Update and Finalize Vectors: These are two lists maintained by the Supernode session. When a particular peer is sent an update message, it is added to the *Update* list. For each peer, which has been updated, the Session Manager expects a confirmation back from the peer indicating that all the changes were made successfully. When a confirmation is received from the peer by the session manager, it removes the peer from the *update* list. Next, it adds the peer to the *Finalize* list. The *Finalize* list consists of all the peers from whom confirmation has been received. When the session manager receives confirmation from all the peers, (indicated by an empty *update* list) it sends each of the peers in the *Finalize* list, the finalize message. Only after receiving the finalize message, the peers start making use of the newly established

connections. By using their old connections till the *Finalize* message is received, the peers ensure that existing participants still continue to receive data while changes are being made. This helps to maintain the content flow even when new nodes join or peers leave the session

4. Recycled ID queue: When a node joins the system, it is assigned an id. This id stays as long as it is part of the system and uniquely identifies the node. When a peer leaves the session, its id is put in the *availableIds* queue. When a new node joins, this queue is checked to see if any of the ids can be reused. If there are no usable ids in the queue then a new id is generated and the node is assigned that id. This prevents the ids from increasing sequentially when there are nodes joining and peers leaving a session continuously.

6.3. SuperNode Design

All of the above classes help form the components of the SuperNode class. The SuperNode is initialized by specifying the port at which the SuperNode listens for requests. It is to this port that all the peers connect to. The port at which the SuperNode listens is provided in the published list of SuperNodes which is made publicly available. The SuperNode consists of a set of Supernode Session objects that are responsible for managing each individual session. The Supernode Session design was described in the previous section. In order to handle multiple requests, the SuperNode class uses multithreading. One thread constantly monitors the listening socket for any incoming connections, while the main thread waits for user input to stop the SuperNode. In addition to this there may be threads spawned to handle requests from peers. These threads exit once they finish handling

the request. Hence, there are 2 threads which are constantly running at the SuperNode. More details about the multithreading aspect is provided later in this chapter. When a request to stop the SuperNode is received, the main thread first cleans up all the data structures it has created and then waits for the monitoring thread to exit before it returns a successful close to the user.

The Figure 6.4 shows the class diagram of the SuperNode. The SuperNode accomplishes its functionality through a list of Supernode session objects, each of which handles a session. The *handleJoinRequest, handleLeaveRequest and handleNodeFail* functions are executed as separate threads which give rise to certain synchronization issues. These issues will be discussed later in this section. Let us go through the components of the SuperNode.

(a) SuperNode Session: As mentioned above, the SuperNode consists of a list of SuperNode session objects, each of which manages a session. All the details regarding a session are present within this object. The SuperNode, when it receives a request, identifies the Supernode session responsible for that session and delegates responsibility to it. A session is identified by an unique sessionId. Each Supernode session object knows the id of the session which it handles. The SuperNode acts as an interface to the peers which connect to it. However the work involved in processing a request is done by the Supernode session.

(b) Multithreaded functions in SuperNode: Multithreading is an important aspect of the SuperNode which helps the SuperNode function effectively and allows some degree of parallelism. The functions that handle join-

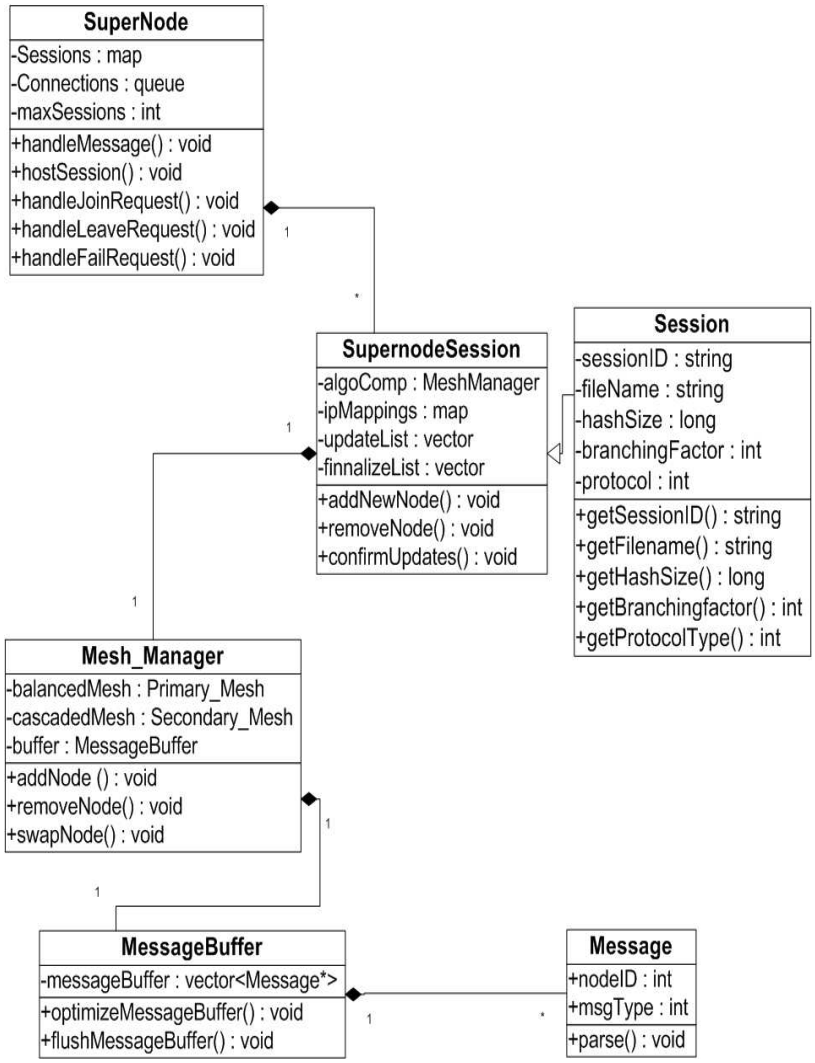


FIGURE 6.4. *Class Diagram for SuperNode*

ing of a node, leaving of a peer or peer failure are executed as separate threads. This helps the monitoring thread to keep listening for more incoming requests. The functions that handle hosting sessions, accept confirmation responses from peers or handle requests to get list of session hosted by the SuperNode, are handled in the monitoring thread as they are relatively smaller in terms of execution time and hence the overhead of spawning a new thread for these purposes can be saved. Besides, requests to list sessions or confirmation of new links from peers in response to *update* messages, can be frequent and hence the frequent spawning of threads is also avoided.

Synchronization issues and their solutions:

Because the SuperNode is multithreaded, race conditions and simultaneous access to common data must be prevented. We briefly go through two major synchronization issues at the SuperNode and see how they are resolved.

i. Prevention of simultaneous data access:

   Problem Description: There is a single mesh in the Session Manager for a session. The mesh is updated in response to both join and leave requests. Join and leave requests are handled in separate threads. If the SuperNode receives a join request and a leave request for the same session simultaneously, then 2 threads are spawned to handle these requests. If both the threads execute simultaneously, then the mesh would be updated by 2 threads simultaneously. As a result the mesh may be in an inconsistent state. To avoid this, the SuperNode has to ensure that only one of these requests is handled immediately and the other request is

kept waiting.

Solution: Simultaneous access to common data (mesh) is avoided by using a mutex. SuperNode assigns a mutex to each ongoing session. When the SuperNode receives a request which is executed in a separate thread, it firsts waits on the mutex assigned to the session that the request refers to. Once the mutex is released, the waiting thread enters the critical section (where the mesh is updated). Next, the thread changes the mesh and *update* messages are sent out to affected peers. The peers then send back confirmation and when confirmation is got from all peers, the *Finalize* messages are sent out. Once all the *Finalize* messages in response to a change are sent, the thread releases the mutex. Thus, the SuperNode does not release the mutex as soon as it is done changing the mesh. It actually waits till it gets confirmation all the peers affected by the change before it releases the mutex.

ii. Handling Race Condition in the Session Manager:

Problem Description: Recall from Section 6.2, the Session Manager maintains the *Update* and *Finalize* lists. The *Update* list contains ids of all updated peers. When we get a confirmation from a peer, the Session Manager moves the peer from *Update* list to *Finalize* list. When the *update* list is empty, the Session Manager assumes all affected peers have sent confirmations and it sends *Finalize* to affected nodes. Consider the multithreaded scenario stated below. The Session Manager in one thread, say thread 1, goes through the Message Buffer and sends updates to the affected peers. These peers are put into the *Update* list. The affected peers test their

new connections and send confirmation to the SuperNode. The SuperNode, in another thread, say thread 2, is listening for confirmation from peers. Once the confirmation from a peer is received, the peer is moved from the *Update* list to the *Finalize* list. So there may be a case where, confirmation from all peers which have been updated is received and the *Update* list becomes empty. Hence the SuperNode sends the *Finalize* message to peers who have sent confirmation. However, thread 1 still continues to send updates to other affected peers. Hence, few of the peers (to which *Finalize* was sent) have made new connections and the others (who received updates after *Finalize* was sent) have not. So the mesh is in an inconsistent state.

Solution: To overcome this problem, when thread 2 detects an empty *update* list, it should know if updates to all affected peers have been sent. Only if updates to all affected peers have been sent, thread 2 should send *Finalize* to all the peers. This is achieved using a flag. This flag is initially reset. The flag is set by thread 1, when updates to all peers have been sent. Thread 2 will monitor this flag and only when it finds that the flag is set and the *Update* list is empty, it will send the *Finalize* message.

Note that both the issues discussed above are applicable only within a session. There are no synchronization issues across sessions since each session is managed by a different Session Manager.

6.4. Object Oriented Principles used in design

The project lent itself to be designed using the object oriented paradigm. The following features were the driving force behind using an Object Oriented Approach in the design for this system.

(a) Project Size: The project was a complex undertaking and had multiple people working on it. Object Oriented Design seemed to be best suited for a project of this size.

(b) Multiple Components: The overall design for the system could be broken down into design of different identifiable components. These components would interact with each other and collectively perform the task at hand.

(c) Object Oriented features: Features like polymorphism, code reusability, encapsulation, etc are extremely useful in developing such a complex system.

Many Object Oriented features were used in the design of the SuperNode. A few of them are listed below.

(a) Object: An Object/Instance is an individual representative of a class. Instances of all the classes that are part of the system are created to perform required functions.

(b) Class: A class is a collection of objects of similar type. Once a class is defined, any number of objects can be created which belong to that class. Examples of classes in the system are *SuperNode, Supernodesession, BMesh*, etc.

(c) Behavior and State: The behavior of a component is the set of actions that a component can perform. A *SuperNode* can handle join request, host request, leave request etc. Its state would be described by the list of Supernode Session objects, queue of connections, etc.

(d) Encapsulation: Storing data and functions in a single unit (class) is encapsulation. This helps to club together the data and all the methods that operate on that data.

(e) Constructors: It is a procedure that is invoked implicitly during the creation of a new object value and guarantees that the newly created object is properly initialized. Every class of the system has a constructor associated with it.

(f) Destructors: Like a constructor, a destructor is a method invoked implicitly when an object goes out of scope or is destroyed. Therefore, just before the object is reclaimed, all resources held by the object need to be released. Every class of the system has a destructor associated with it.

(g) Inheritance: It can be defined as the process whereby one class/object acquires (gets, receives) characteristics from one or more other classes/objects. The class from which the properties are inherited is called the parent class and the class that inherits the properties is called the child class. The child class obtains all the variables and methods from the parent class and being a more specialized form of the parent class, it adds additional functionalities. In the SuperNode design, inheritance is used for a particular purpose as explained below.

i. Subclassing for Specialization: The inheritance relationship between *Message* class and all classes derived from it exhibits this form of inheritance. Classes like *ackMessage, connMessage, etc* are specialized from the *Message* class. The child classes do not provide any additional functionality but modify behavior like *makeMsg, parse* to behave according to the message they represent.

ii. Subclassing for Extension: The inheritance relationship between *Session* and *SupernodeSession* classes exhibits this form of inheritance. In this case the *SupernodeSession* simply adds new methods to the *Session* class and provides it with new abilities. The *SupernodeSession* class has the ability to maintain the logical view of an entire session and also add or remove nodes from the session.

(h) Polymorphism: The term polymorphism means many forms (poly = many, morphos = form). Polymorphism in programming languages means that there is one name (function or method or variable or class name) and their meanings can be defined in a number of different ways. Polymorphism is exhibited in the following ways at the SuperNode.

i. Overriding (inclusion polymorphism): In case of Message subsystem, classes which inherit from the *Message* class, have the *makeMsg* and *parse* functions which are inherited from the base class. These classes override the definition provided in the parent class.

ii. Polymorphic variable: The Message Buffer is a collection of *Message* class objects. However this collection never holds *Message*

class objects. It can hold objects of any class derived from *Message*. Hence the *Message Buffer* is a polymorphic variable.

iii. Overloading: If we consider the different classes inherited from *Message*, they exhibit *Overloading based on scope* as there are functions with the same signature that are present in all these classes but they are at different scopes. Hence the same function name takes multiple forms depending on scope.

(i) Composition: The *SuperNode* has instances of the *Supernode session* and an instance of *Message Buffer* as a data field. Similarly there is composition even inside the *Supernode session* and also different components of the *Mesh Manager*. These objects help each other and collectively perform the required functions.

(j) Containers: *Vector* is an array of objects whose size can be dynamically changed. Vectors are used extensively in this system. For example they are used to store the *update* and *Finalize* lists for a *SuperNode session*.

(k) STL entities such as *map* and *Vector* have been employed extensively in this system.

(l) Design Patterns: A pattern is an attempt to document a proven solution to a problem so that future problems can be more easily handled in a similar fashion.

i. Iterator: The Iterator is the pattern that is visible in the implementation of the system. It is used in order to provide access to the elements in a collection (vectors) without exposing the internal organization of the container.

ii. Singleton: The SuperNode class is a singleton as there can be only one instance of the SuperNode running for a given execution.

iii. Proxy: The Message class hierarchy hides the protocol details from the rest of the system. The *Peer* and *SuperNode* communicate with each other using this *Message* subsystem but both these classes are unaware of the exact message formats..

7. SUPERNODE WORKING

The SuperNode is the part of the system that is constantly working. It monitors the listening port for any incoming requests from peers and handles these requests. In this chapter, we study how the SuperNode responds to various requests it receives from peers and also see how mesh optimization is done by the SuperNode.

7.1. Handling list session requests:

This is a request from a node, for the list of sessions handled by the SuperNode. For each session that the SuperNode handles, the SuperNode gives session details like what file is being streamed, what is the protocol, etc. When the SuperNode gets such a list session message, it is handled by the monitoring thread itself. The list session response is generated by fetching the session details for each active Supernode session. The SuperNode then sends the response back to the requesting node.

7.2. Handling host session request:

When a peer has a file that it wants to stream, it will send a host session request to the SuperNode. The request contains details like the file name, file size, protocol used for streaming, etc. The SuperNode notes these details and provides it when any other node requests session details. When the SuperNode receives a request to host a session, it will determines how many sessions it is currently handling and checks if it can accommodate

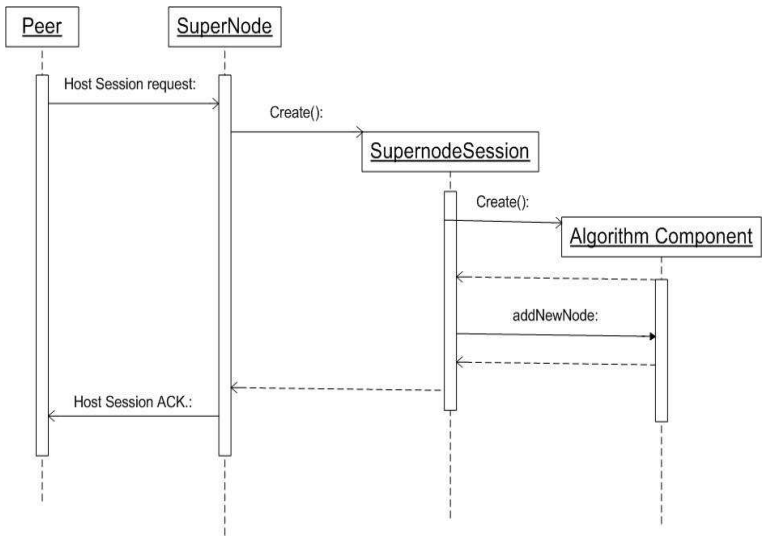


FIGURE 7.1. *Sequence of events for successful session hosting*

another session. If it can accommodate another session, it will create a new Supernode session object. This Supernode session will be responsible for the newly hosted session. It will then send an Host Session ACK along with the id of the hosted session, to the peer. If the SuperNode cannot host another session, then it will send a NACK to the peer indicating the server is full. The Figure 7.1 shows the sequence of events that take place at the SuperNode in case of a successful session hosting.

7.3. Handling join session request:

When a node intends to join a session, it sends the join message to the SuperNode. This message contains the id of the session the node wants to join, the node's control port and data port, and its upload capacity (which will be used for optimization). When the SuperNode receives the request,

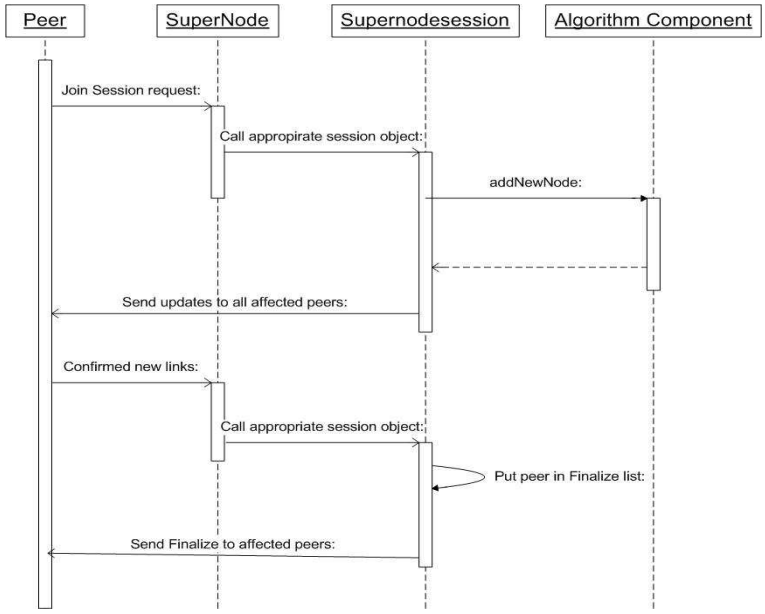


FIGURE 7.2. *Sequence of events at the SuperNode for a join request*

it looks up the Session Manager for the session the node intends to join and invokes the *addNode* function on that session. In the *addNode* function, an id is generated for the node. As described in Chapter 6, the ids may be recycled or sequentially generated. Once an id is generated for the node, an entry is made into the *mappings* table. This entry maps the node's id to node details (IP, control port, data port). The mapping is important as the Session Manager is the only place where these ids can be translated into IPs and port numbers. After the mapping entry is made, the Supernode Session calls the *addNode* function of the Mesh Manager which runs the algorithm and adds the new node to the mesh. The resulting set of messages are stored in the Message Buffer. Now, the Supernode Session goes through the buffer and sends update messages to affected peers. The new node receives a set

of peers as its neighbors and sends its confirmation to the SuperNode. After the Supernode session has received confirmation from all the affected peers, it sends the *Finalize* message. Hereafter, the new node is part of the ongoing session. The Figure 7.2 shows the sequence of events that take place at the SuperNode in case of a join request.

7.4. Handling peer leave requests or peer fail messages:

For both peer leave and peer fail, the peer must be removed from the mesh. Hence the handling of leave requests or fail messages is similar. When a peer intends to leave a session, it will send a leave message to the SuperNode. In case of a failed peer, a message is sent by one the neighbors of the peer which is expecting data from it. For peer fail, the reporting peer also sends out the IP of the failed peer. Once the leave or fail message is received, the SuperNode will invoke the appropriate session object and call *removeNode* on that object. The session object in turn calls the *removeNode* of the Mesh Manager. The Mesh Manger runs the algorithm and populates the Message Buffer and the Supernode session sends out the updates. Once all the affected peers confirm their changes, the Supernode session removes the peer from the mapping and also returns its id to the queue of available ids. In case of a leave request, the session manager also sends out an ACK to the leaving peer. Only after receiving this ACK, the peer actually leaves the system. Till then it is part of the system and helps maintain continuous flow of data. The Figure 7.3 shows the sequence of events that take place at the SuperNode in case of a leave request from the peer.

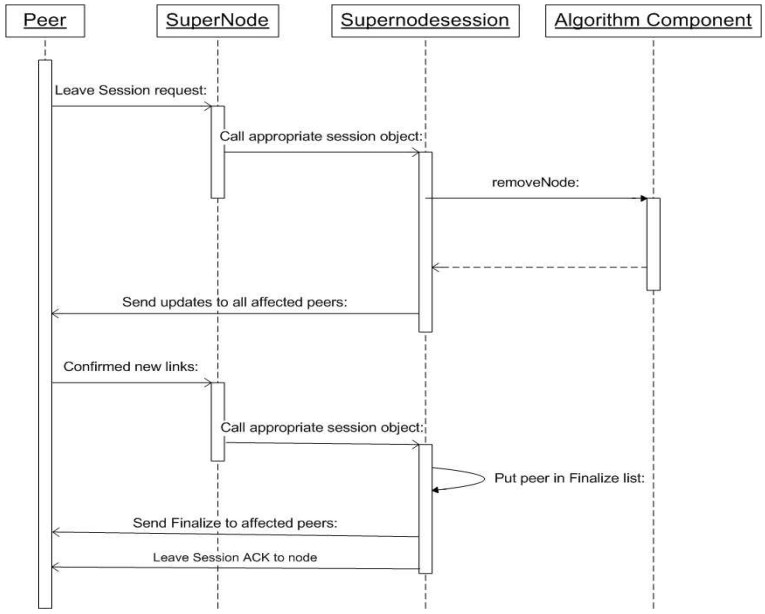
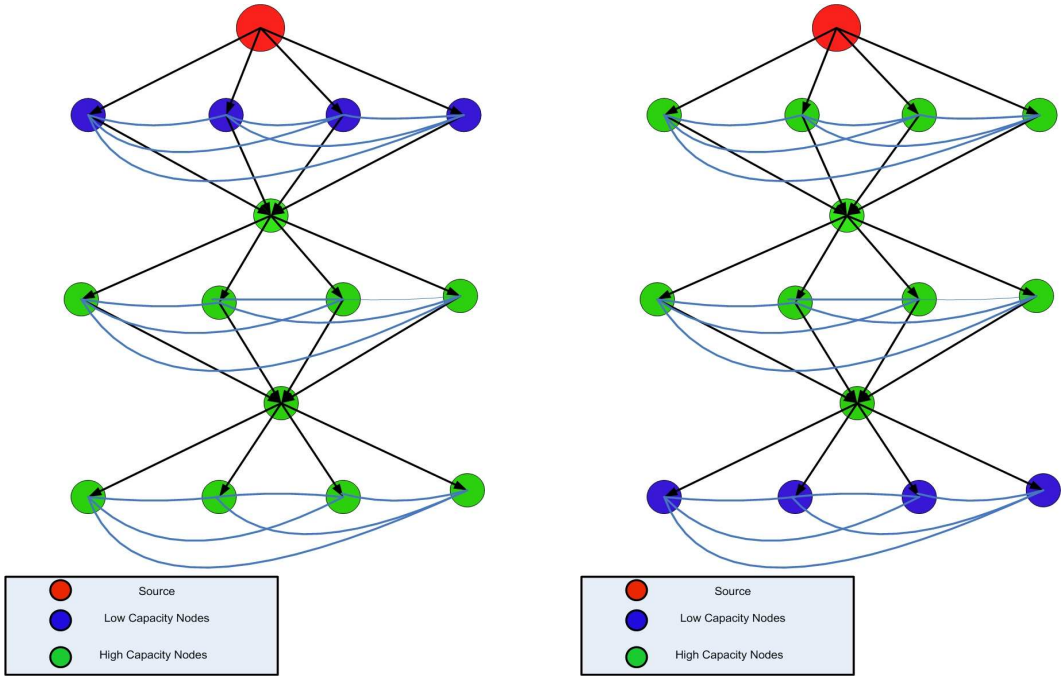


FIGURE 7.3. *Sequence of events at the SuperNode for a leave request*

7.5. Mesh Optimization

Without optimization, the Mesh Manager assumes that the upload capacity of all the peers in the system is similar and hence the positions of the peers in the mesh are not important. However when the upload capacity of peers varies then it becomes imperative to come up with a scheme which ensures maximum throughput possible for the given set of peers. Clearly if the low upload capacity peers are closer to the root of the mesh, they are bound to affect more peers than if they were placed at the leaf level. The leaf peer affects, the peer to which it has cross links, the parent of the cross link (if the cross linked peer has a back link) and its own parent to whom it forwards data. The idea behind mesh optimization is to keep all the low capacity peers at the greatest depth possible.



(a) (b)

FIGURE 7.4. *(a) 15 peer topology for $b = 4$ without optimization (b) 15 peer topology for $b = 4$ with optimization*

The capacity of a peer is determined at join time. Recall that the peer sends a join session message which contains the peers capacity. This can be one of T1, DSL or DialUP in decreasing order of upload capacity. The capacity of every node is compared with its parent's capacity (except the secondary root, for which the capacity of its reference is considered) and if the parent is not a T1 peer, the parent is added to a queue, *swapCandidates*. In the optimized version of the Mesh Manager, when a peer joins, it is first added to the mesh. Next, the Mesh Manager goes through the *swapCandidates* queue to determine if there is any lower capacity peer in the queue. If such a peer exists, then the incoming peer is swapped with the low capacity peer. Since the data structure used is a queue, swap candidates are identified in level order fashion which ensures that if there is any low capacity peer higher up in the mesh then it is swapped. Also, if the peer (say A) which replaces the low capacity peer (say B) is not a T1 peer (in case a DSL has replaced a DialUP) then peer A takes the place of peer B in the *swapCandidate* queue. This ensures that if a better capacity peer (say T1) joins in later, then peer A gets swapped. The Figure 7.4 shows the effect of swapping. In this case, the first four peers which were added to the mesh were low capacity followed by all high capacity peers. In Figure 7.4(a) all the low capacity peers remain closer to the root and hence all the peers below them are affected. In Figure 7.4(b) due to optimization, all the low capacity peers are swapped down to the greatest depth and hence affect lesser peers.

8. PERFORMANCE EVALUATION

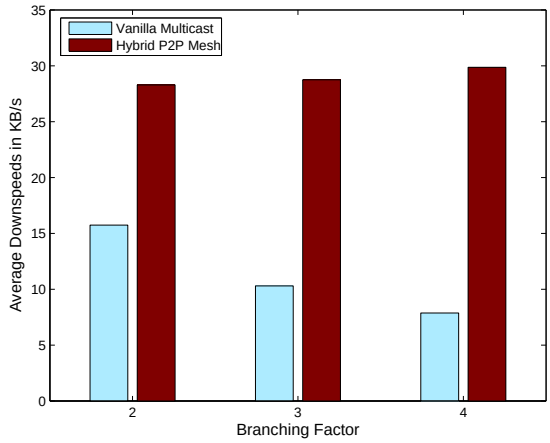
8.1. Small Scale Experiments

The P2P system developed was deployed on PlanetLab [25] machines. An extensive set of experiments were run, the results of which are depicted and discussed below.

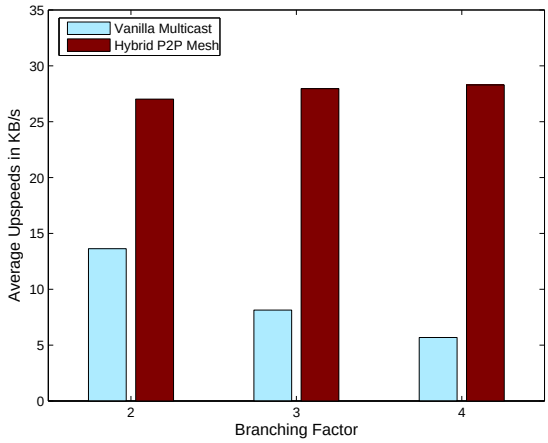
8.1.1. System Throughput Evaluation

The following experiment was carried out to compare the throughput of our system against Vanilla Multicast. As we saw in section 1.2.3, in case of Vanilla Multicast, the onus of forwarding data was completely on the internal peers and the leaf peers had a lot of unused bandwidth. The experiments were run on the same set of peers for both Vanilla Multicast and our system. Also to get an accurate and unbiased result, we kept the logical position (position in the topology) of machines the same. To simulate the DSL upload bandwidth bottleneck, the sending rate of the source was limited to 30kBps

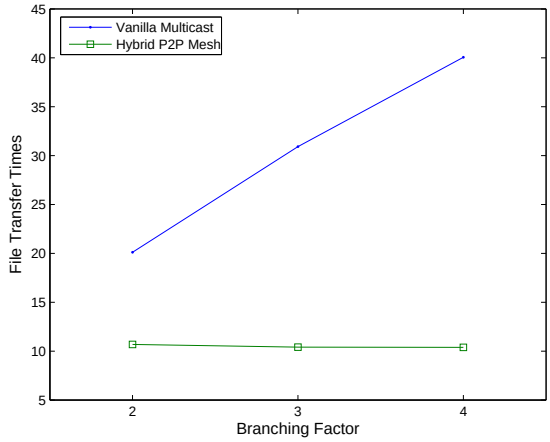
The result of the experiment is shown in Figure 8.1. As seen, the proposed system outperforms Vanilla multicast in each experiment. Further as the value of branching factor increases, the performance gain of the proposed system increases. In theory the proposed system outperforms Vanilla multicast by a factor of b. This is because as the value of b increases there are more leaf peers than internal peers in a tree. In case of Vanilla multicast, only the internal peers upload data and hence the total upload capacity of the system decreases. This can be seen in Figure 8.1(b). The proposed system utilizes the leaf node bandwidth. Even when



(a)



(b)



(c)

FIGURE 8.1. *(a) Average Down Speeds for a 15-Peer Topology (b) Average Up Speeds for a 15-Peer Topology (c) Comparison of file transfer time for a 15-Peer topology*

there is an increase in branching factor b, the topology is such that each node still has b incoming connections each of which delivers content at the rate of C/b and hence system performance does not degrade with increase in b. As seen from Figure 8.1, the upspeed and downspeed for the proposed system is almost the same for any value of branching factor. This is in accordance to the notion that for the proposed system throughput efficiency is 1. The Figure 8.1(c) compares the average file transfer time for Vanilla multicast and the proposed mesh. For this experiment, a file of size 315 kB was sent by the source. The transfer time for Vanilla multicast increases with increase in b. This is because with increase in b the average upload speed decreases in Vanilla multicast. But, for the proposed mesh, the transfer time remains constant for different values of b which is a result of constant upload speeds for different values of b.

8.1.2. Packet Delay Evaluation

These set of experiments were aimed to measure the time taken for a peer to receive the first set of data from all the groups. We know from Section 3.3 that the source partitions the data and transmits different partitions through different branches. A peer is said to have received complete data only after it receives data from all the groups. The data from various groups follow different paths along the topology and are also routed individually across the network by the routers. Hence there is no particular order in which packets can be expected.

Figure 8.2 shows the result of the packet delay experiment. As expected the delay decreases with increase in branching factor. This is because with increase in b the depth of the mesh decreases and hence there are lesser number of hops to get from the source to the destination. Note that as b increases there are more

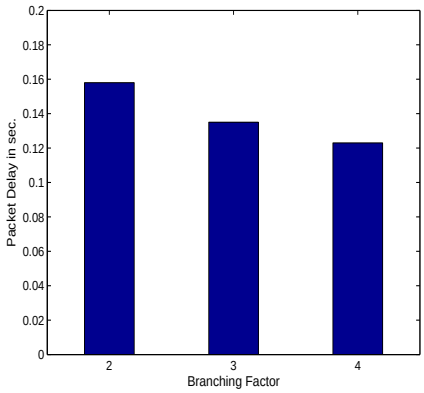


FIGURE 8.2. *Average Packet Delay for a 21-Peer topology*

number of cascaded meshes in the secondary mesh and hence the delay for these peers might increase. But in most practical cases, the number of peers in the primary tree is greater than that in the secondary tree. For $b = 3$, the number of peers in the primary mesh is 13 while there are only 8 peers in the secondary mesh. Hence the *average* delay decreases.

8.1.3. Peer Join Evaluation

In the next set of experiments, we compare the join times and the mesh management overhead for different topologies with out-degree set to $b = 2$, $b = 3$ and $b = 4$. Before evaluating the results we would like to define certain logical points in the algorithm. These logical points refer to the state of the mesh when the node joins in and hence determines the join time. They are as follows:

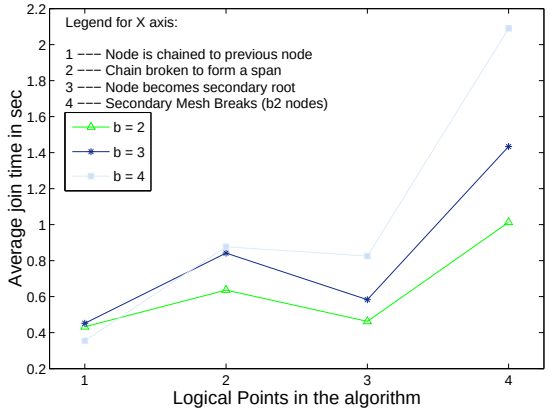
1. Node is chained: In this case, the incoming node is connected to only the last peer in the mesh which forwards it data from all the groups.

2. Chain is broken to form span: This occurs for nodes which become part of the secondary mesh. In this case, the secondary mesh has a set of peers chained. This incoming node makes the node count $b + 1$ which results in the change from chain to span topology. Figure 6.2 shows the chain and span topology in the secondary mesh.

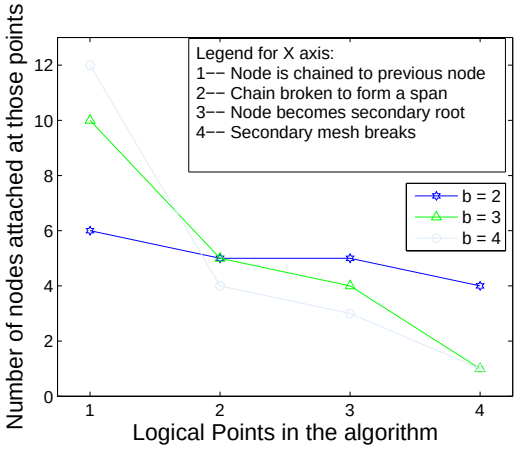
3. Node becomes secondary root: In this case, there is a balanced primary/secondary mesh and the node coming in becomes the root of the new secondary mesh.

4. Secondary Mesh breaks: This is the point in the algorithm where the incoming node results in b^2 nodes in the secondary mesh. As a result the secondary mesh breaks and is reattached to the primary mesh.

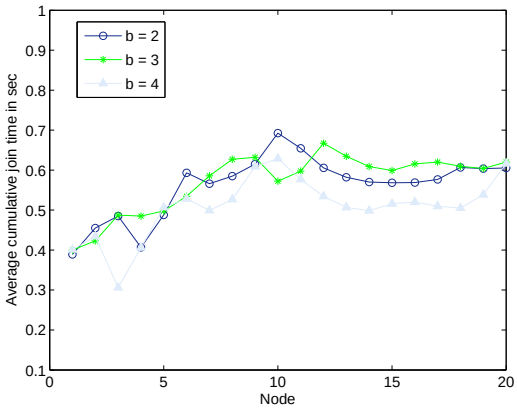
The Figure 8.3(a) shows the join time for peers at different logical points in the algorithm. We observe that, as the value of b increases, the average join time at different logical points 2, 3 and 4 increases. For point 1, the peer is chained to the previous peer and hence the join time does not depend upon b but depends upon the proximity of the 2 peers to each other and to the SuperNode. At points 2, 3 and 4 the number of peers affected and hence the join time, depends upon b. As b increases, the number of peers affected increases and hence join time increases. If we study the graphs for $b = 2, 3$ and 4 indiviually, it is observed that all 3 topologies follow a similar pattern. The peer being chained requires the least time. When the chain is broken and when a peer becomes a secondary root, totally $b+1$ peers are affected. But in the former case, an existing chain is broken to form a span, while in the latter case, b peers directly connect to the new peer. Hence the join time for the former is more than that for the latter. Finally at



(a)



(b)



(c)

FIGURE 8.3. *(a) Average join times at different points in the algorithm for 21-Peer topology (b) Number of peers joining at different logical positions for 21-Peer topology (c) Cumulative average join time for 21-Peer topology*

point 4, the secondary mesh breaks and hence the number of peers affected is of $O(b^2)$ and as b increases, the join time at this point increases sharply.

Figure 8.3(c) shows the cumulative average join times for different values of b. As expected, initially the graph changes rapidly with each incoming peer, before becoming stable. It can be seen from the graphs that the average join times for all values of b is approximately the same. The average join time is determined by the number of peers joining in and their logical positions in the algorithm. Figure 8.3(b) shows the number of peers that join at each logical position. When a peer joins and is chained, the join time is very small. Whereas, when the mesh is broken, the join time is comparatively large. As seen in Figure 8.3(a), when the mesh breaks, the join time at point 4 for $b = 4$ is much larger than for $b = 2$. However for $b = 2$, the mesh breaks for every 4 peers while for $b = 4$, the mesh breaks only every 16 peers and hence the average join times for different values of b is similar. It can be seen from Figure 8.3(b) that most of the peers for $b = 4$ are chained which really contributes towards decreasing the average join time. In fact as shown in Figure 8.3(c), the average join time till peer 19 is lesser for $b = 4$ than for $b = 3$ or $b = 2$. But at peer 20, the secondary mesh in $b = 4$ breaks and hence there is a spike in the average join time and the average join times for all bs are similar. Similarly in Figure 8.3(c), for the graphs of $b = 2$ and $b = 3$, there are spikes corresponding to breaking of the secondary mesh (peers 6, 10 and 18 for $b = 2$ and peer 12 for $b = 3$).

Data flow at SuperNode for Join requests:

The Figure 8.4 shows data flow at the SuperNode in response to join requests it receives. This can be considered as the *Mesh Management Overhead*. As expected, the data flow increases with b. This is because as b increases, the

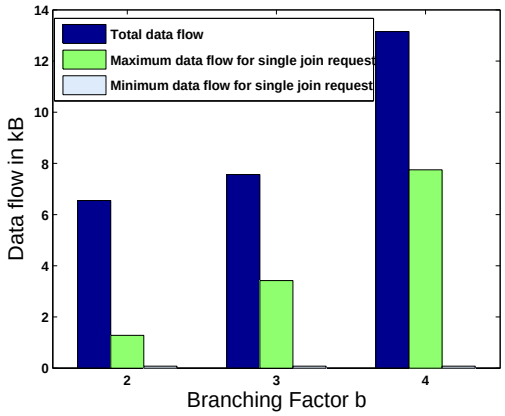


FIGURE 8.4. *Data flow at SuperNode for join requests for 21-Peer topology*

number of affected peers, to whom messages are sent also increases. The minimum data flow occurs when a peer is chained. Since the number of affected peers in this case is constant for all bs, the minimum data flow is similar for all values of b. The maximum data flow in each case occurs when the mesh is broken and hence as b increases, the maximum data flow increases rapidly ($O(b^2)$). As the value of b increases, it can be seen that the contribution of the maximum data flow to the total data flow increases as well. Since this experiment was run for 21 peers, for $b = 2$, the mesh was broken multiple times while for $b = 3$ and $b = 4$, the mesh was broken only once. Hence, the maximum data flow contributes significantly to the total data flow for $b = 3$ and $b = 4$. However, when there are more peers involved, the total data flow keeps increasing while the maximum data flow for a single join remains constant. Thus, the system scales up well in presence of more peers.

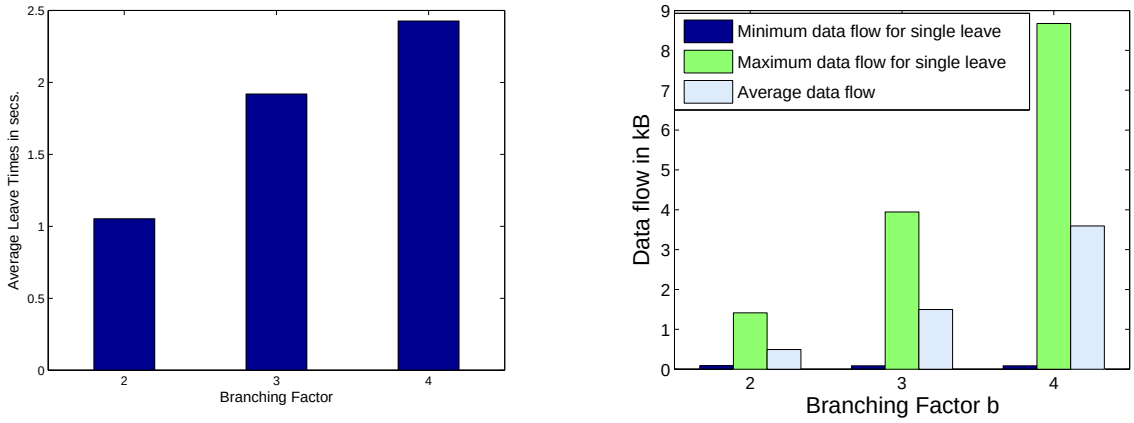
8.1.4. Peer Leave Evaluation

This set of experiments try to evaluate the time taken by a peer to leave a session and the overhead of the leave at the SuperNode in terms of messages generated. The experiment was run for a 21 peer topology and for branching factors $b = 2, 3, 4$. There are various scenarios in which a peer can leave and each scenario is handled separately. In order to make sure that all the scenarios were assessed before calculating an average leave time, we ran the experiment for the following cases:

1. Secondary Mesh is not present and Primary Mesh is balanced
   (a) A leaf peer leaves.
   (b) A peer high in the hierarchy leaves.
   (c) A random peer leaves
2. Secondary Meshes are present
   (a) A peer high in the primary mesh leaves.
   (b) A leaf peer of the secondary mesh leaves.
   (c) A chained peer in the secondary mesh leaves.
   (d) The root of the non-empty secondary mesh leaves.
   (e) The root of the empty secondary mesh leaves.

The Figure 8.5 shows the result of the experiment.

Clearly as the value of branching factor increases, the leave times also increase. The main contribution to the leave time is mesh reconstruction after the leave which involves sending messages to all the affected peers. As discussed



(a) (b)

FIGURE 8.5. *(a) Average leave times for a 21-Peer topology (b) Data flow at SuperNode for leave requests for a 21-Peer topology*

in Section 3.5.2, we know that the number of affected peers for a leave is at most $b^2 + 2b$. This number increases rapidly with the increase in b.

Data flow at SuperNode in response to leave request:

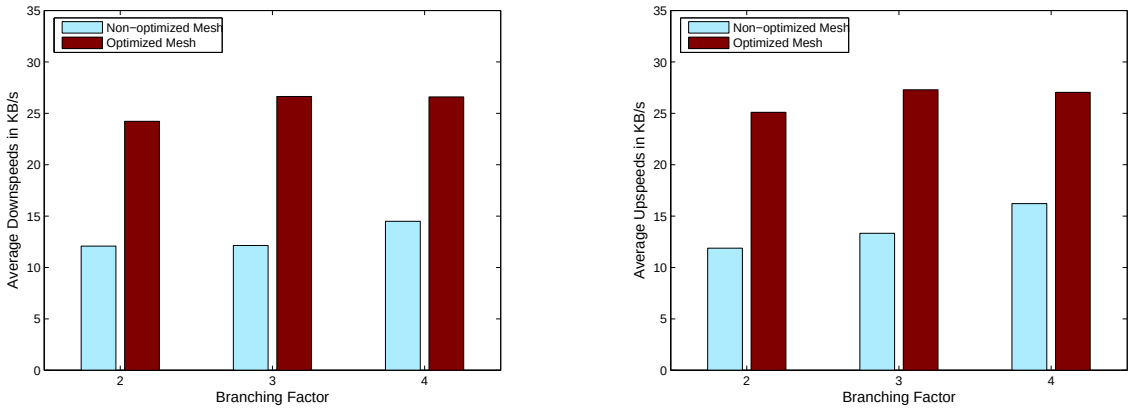
Figure 8.5(b) shows the overhead at the SuperNode when a peer leaves the mesh. Similar to the join overhead, the average data flow increases with increase in b. Again the minimum overhead is incurred when a chained peer leaves. In this case, the only peer affected is its parent and hence the data flow is constant for different values of b. The maximum data flow occurs when an internal (non leaf) peer leaves a perfectly balanced mesh. In that case, first a leaf peer is swapped with the internal peer and then out of the remaining $b^2 - 1$ peers, a secondary mesh is formed. This process involves first sending out a set of messages to swap in a new peer in place of the leaving peer, then disconnection messages to break off the lowermost level from the primary mesh and lastly a set of connect messages

to construct the secondary mesh. As b increases, the number of nodes required to form a secondary mesh increases and hence the data flow increases. On an average, when a peer leaves the number of peers affected increases with b and hence the average data flow in response to leave requests increases with b.

8.1.5. System Throughput Evaluation of an Optimized Mesh

The next set of experiments evaluate the performance of the optimized mesh against the non optimized mesh. The non optimized mesh is worst hit when the initial peers which join the session are low capacity peers. Hence to achieve maximum performance gain due to optimization, in the experiment carried out, the first 3 peers that were added to the mesh were ones with low capacity. These were set as DialUp which were followed by T1 peers which had higher capacities. We again carried out the experiment for $b = 2, 3, 4$. The experiment for system throughput was run on the local network with the sending rates throttled for peers marked as DialUp.

Figure 8.6 shows the result of the experiment. As expected the optimized mesh results in greater throughput. This is because when optimization is enabled, the Mesh Manager swaps the low capacity peers to the bottom while the high capacity peers move up the mesh. If there is no optimization then there is no swapping of peers and hence the first three low capacity peers stay close to the root of the mesh and affect all subsequent incoming peers. When the low capacity peers are at a greater depth (optimized mesh) they are bound to affect lesser peers than they would if they were at the top of the hierarchy (non optimized mesh). Figure 7.4 depicts meshes with optimization and without optimization.

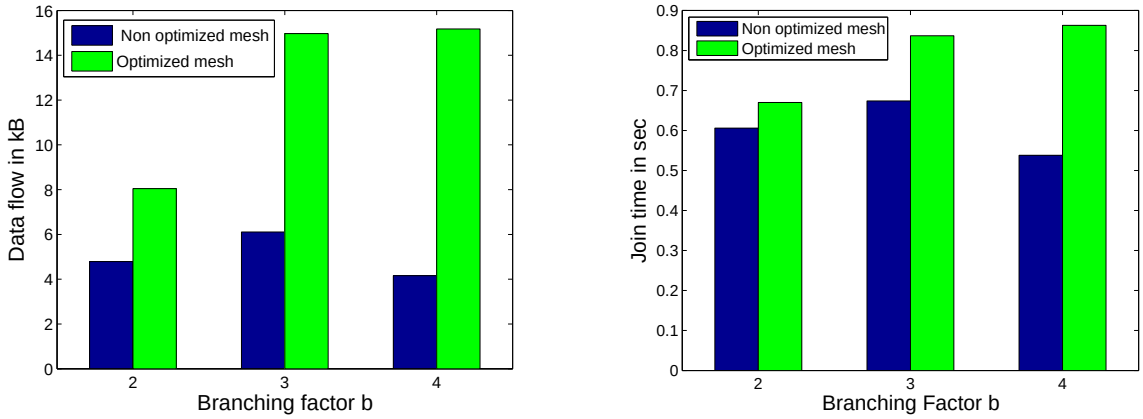


(a) (b)

FIGURE 8.6. *(a) Average Down Speeds for a 15-Peer Topology (b) Average Up Speeds for a 15-Peer Topology*

Overheads involved in mesh optimization: When a peer joins into a mesh in which optimization is enabled, the SuperNode will determine the peers capacity and see if there is a lower capacity peer higher up in the mesh. If such a peer exists, the SuperNode will swap the new peer with the low capacity peer. As a result, the data flow during join in an optimized mesh is more than that for a non-optimized mesh in which no swapping occurs. Along with increase in data flow, there is also increase in the join time of peers.

The figure 8.7(a) shows the comparison of data flow for join for non optimized and optimized meshes for different values of b. It is evident that the data flow for optimized mesh is greater than that for non optimized mesh. Recall that the first three peers added to the mesh were of low capacity and the rest T1. Hence with each subsequent T1 peer addition, there will be a swap with a low capacity peer closer to the root. In the final state, the low capacity peers are at



(a) (b)

FIGURE 8.7. *(a) Total data flow for join for 15-Peer topology (b) Average join time for 15-Peer Topology*

the bottom of the mesh. Since the swap is performed for almost every peer added, the data flow overhead is significant. Also for $b = 4$, for a 15-peer topology, a lot of peers get chained and hence the non optimized data flow is very small. But when optimization is enabled, these peers which were originally chained now get swapped with other peers higher up generating more messages. Hence the overhead is maximum for $b = 4$. The overhead is lesser for $b = 2$ where the mesh gets broken often and hence a lot of data is generated even for the non optimized mesh. In this case, swapping overhead in terms of messages generated is insignificant. Figure 8.7(b) shows the comparison of average join times for non optimized and optimized meshes for different values of b. The results are similar to that for data flow. For $b = 2$, the join time in case of the optimized mesh is similar to that for the non optimized mesh. But for $b = 4$, since most of the chained peers get swapped, the join time increases significantly.

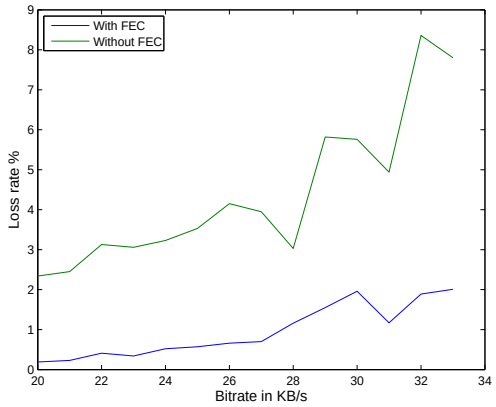


FIGURE 8.8. *Packet Loss Evaluation*

8.1.6. Packet Loss Evaluation

Low loss rate is desirable for media dissemination to ensure quality of service. With peer-to-peer architecture, the data transmission rate is more unpredictable since peers participate in data forwarding. An unreliable peer will affect data transmission of its immediate peers. We conduct an experiment on PlanetLab to measure the packet loss rate of the system. The experiment consists of 7 peers, branching factor b = 2, and packet size = 500 bytes. In addition, we implemented forward error correction (FEC) with 30 % redundancy. Figure 8.8 shows the effect of increase in sending bit rate, on loss rate. As seen, as the sending rate increases, the loss rate increases. For each bit rate, we ran two different sessions. A session's loss rate will be the average of the loss rates of all of the receiving peers inside the mesh. The loss rate plotted in Figure 8.8 is the average of these two sessions. Loss rate occasionally decreases due to improved traffic conditions on PlanetLab. However, the overall loss rate tends to increase as bit rate increases. As seen, FEC does help reduce loss rate. For example, at a

sending rate of 33 Kbytes/sec, loss rate with FEC is about 2.0% while loss rate without FEC is about 7.8%."

8.2. Large Scale Simulation

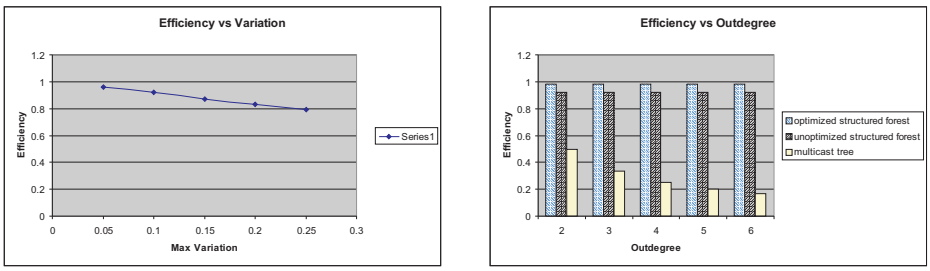
8.2.1. Throughput Efficiency

In this simulation we use 3000 nodes with capacities uniformly generated between $C(1+v)$ and $C(1-v)$ where C is the mean capacity. Figure 8.9(a) shows the throughput efficiency for our structured mesh vs. maximum variation on capacity v. As seen, the efficiency reduces as the capacity variation increases since an internal node may have small capacity which creates a bandwidth bottleneck for all its children. However, even when $v = 0.25$, the throughput efficiency is still 0.8%. Similar results are obtained when node capacity is normally distributed.

Figure 8.9(b) shows the throughput efficiency vs. the out-degree for three different schemes: traditional multicast tree, unoptimized structured mesh, and optimized structured mesh. For optimized structured mesh, nodes with lower capacities are moved to the leaves to reduce bottleneck for other nodes. As seen, throughput efficiency is 98% for optimized structured mesh and 92% for unoptimized one. For the multicast tree, the throughput efficiency is small and decreases as the out-degree increases since the number of inactive nodes (leaf nodes) increases in this topology.

8.2.2. Robustness Evaluation

We now present the simulation results for our proposed structured mesh which looks into the robustness of the system. The following simulations aims



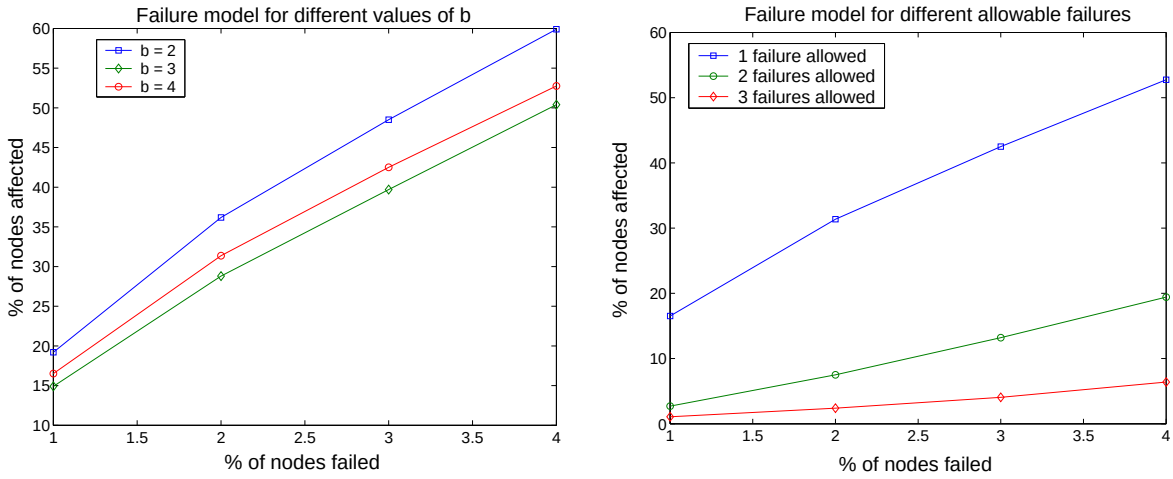
(a) (b)

FIGURE 8.9. *(a) Efficiency vs. variation; (b) Efficiency vs. out-degree for different data dissemination schemes.*

to quantify the affect of node failure on the proposed topology. All simulation was done using NS [9]. In this simulation, we used BRITE [10] to generate an *Albert-Barabasi* topology consisting of 1500 routers. Next, we randomly generated an additional 1000 overlay nodes and connected them to the existing 1500 routers. There are two important factors that determine how the failure of a node affects others in the topology. The first is whether the node in contention is a leaf node or an internal node and the second is the branching factor b. If the node is an internal node, then when it fails, the number of affected nodes are more than that in the case of a leaf node. This is because all the nodes below that internal node and the nodes in the other groups that receive data due to cross links also fail. The branching factor determines how many nodes a particular node is connected to and hence, provides data to. Hence, larger the branching factor, more the number of nodes affected for a given failed node.

Figure 8.10(a) shows the percentage of affected nodes as a function of failed nodes for different branching factors. It is important to emphasize that these failures are only temporary as the network can reconstruct itself as described in

Section 3.5.2. As expected, the percentage of affected nodes increases with the percentage of failed nodes. For $b = 2$, the number of internal nodes is large (500 nodes) and hence the number of affected nodes is largest. It is interesting to note that for $b = 3$, the number of non leaf nodes is 336 and for $b = 4$, it is only 254. Between $b = 3$ and $b = 4$, the difference in the number of internal nodes is not large, but because of the branching factor, the affected nodes for $b = 4$ is higher than $b = 3$.



(a) (b)

FIGURE 8.10. *(a) Percentage of affected nodes as a function of percentage of failed nodes for different values of branching factor b; (b) Percentage of affected nodes as a function of percentage of failed nodes for different allowable failures with* $b = 4$.

Let us suppose FEC or Multiple Description Coding technique is used to disseminate the data [11] [12]. In this case, a node need not receive complete data. Thus, a node is considered a failed node if it fails to receive more than a certain number of partitions K. Figure 8.10(b) shows the percentage of affected nodes as a function of percentage of failed nodes for different K with $b = 4$. As

expected, the number of affected nodes decreases as more packet loss is allowed. The reduction in percentage of affected nodes is significant. This is because the data received at a node is coming from different parts of the topology and so it will require a large number of nodes to fail to completely deprive a node of any data.

9. RELATED WORK

In this section, we look at similar work on data dissemination on the Internet.

Similar work is done by [15] in developing *Bullet*, a high bandwidth data dissemination system. They make use of an overlay mesh at the application layer. In case of *Bullet*, the source splits the data into disjoint parts and sends them to the peers. It is then upto the peers to find out what part of data they are missing and which peers have them. A peer then establishes multiple connections to other peers having data disjoint from them and hence gets complete information. This work makes use of data reconciliation techniques like *minwise sketches* described in [13] to locate disjoint data. Byers et.al. [13] suggests a number of useful ways to locate content in an overlay network. They provide techniques of data reconciliation at different levels. They go into details of coarse grained reconciliation, speculative reconciliation and fine grained reconciliation any of which can used depending on the disjointness of data between two content seeking entities. They address the issues of heterogenous connections and asynchrony by advocating the use of encoded data.

In both the cases cited above, the stress is more on locating content among the peers once it has been disseminated by the source. None of these works focus on algorithmically creating a topology that will obviate the need for such dynamic information exchange. Our work tries to construct a good topology in which nodes connect to each other in a deterministic manner as instructed by a controlling authority. Hence there is no need for data reconciliation.

Authors in [11] use multiple overlay multicast trees to achieve the objective of video streaming. In this case they transmit multiple descriptions of the video

along different multicast trees and these multicast trees co-operate to ensure peers get as many descriptions as possible. The more descriptions a peer gets, the better the playback quality of the video. The technique of multiple multicast trees to disseminate data is also used by SplitStream [16]. However in SplitStream, care is taken that a peer is an internal node in only one of the multicast trees and is a leaf in the rest. This ensures that the system is reliable and the peers are deprived of only one description (the description sent out on the tree in which it is an internal node) if another peer fails.

In most of the above mentioned schemes, there is no algorithm involved in mesh construction. In case of *Bullet* the onus is on the peer to locate content it wants while SplitStream makes use of Scribe [17] and Pastry [18] infrastructure for tree construction. This thesis focusses on a deterministic algorithm that specifies the manner in which nodes are to be attached for optimal data dissemination.

The most similar work to that presented in this thesis has been carried out by MutualCast [3]. Similar to our approach, MutualCast looks to utilize the upload bandwidth of all the nodes. MutualCast uses the bandwidth of nodes that request the content and in addition also uses certain non content requesting nodes to forward data. MutualCast constructs a number of height-1 (where the source is directly connected to all the content requesting nodes) or height-2 (where the source connects to an intermediate content requesting or non content requesting node) trees that aid the process of content dissemination. The intermediate nodes in the height-2 trees have an out degree of O(N1) where N1 is the number of content requesting nodes. Hence the number of nodes affected for a join or leave will be of O(N) where N is the total number of nodes. In this scheme the number of outbound connections for the source is of O(N). In case of MutualCast, the onus is on the source to decide whether to forward a data partition directly to the

destinations or to send it down a height-2 tree and if it opts for the latter, it will have to decide which tree to send the data partition to.

Similar work include [19] which proposes a protocol for cooperative bulk data transfer. Other similar work try to reduce the burden on the server by utilizing the upload bandwidth of the peers. These schemes become inherently more scalable, as more the number of peers, more the upload capacity of the system. For example, authors in [20] make use of P2P overlay networks formed by the clients themselves to alleviate the traffic burden on the content servers. The capacity modeling of P2P file sharing systems have also been studied by [21] [22].

Real world applications similar to ours include P2P networks like Gnutella [6], KaZaA [7], Swarmcast [23], and BitTorrents [24]. These systems also utilize peer capacities to distribute content to multiple receivers in a cooperative environment. In case of BitTorrents, a client can start uploading a file which it has not completely received. In this case, the peer is simultaneously receiving data and uploading data belonging to the same file transfer session.

10. FUTURE WORK

The system in place is an experimental system and is by no means complete to be deployed and used as a regular application. Since the system was mainly created as a proof of concept, there are a few features which can be implemented to take this system to the next level of usability. As mentioned in Section 4.2, the system already has in place features like Heartbeats and Mesh Optimization. In addition to these, the following features could also be incorporated into the system.

1. Distributed SuperNodes: In the current system, there is a single SuperNode which is responsible for handling all details regarding a single session. Hence this SuperNode acts as a single point of failure. The failure of the SuperNode leaves the session in a dangled state. No nodes can join the session and also the source cannot close the session as the SuperNode handling the session is unavailable. This drawback can be removed by having multiple SuperNodes hold information regarding the same session. Also, all peers part of the session could be given a list of SuperNodes in charge of the session. Then, the SuperNode is no longer a centralized point of failure.

2. Dynamic Mesh Optimization: Presently, mesh optimization is based on static information about capacity provided by the peers at join time. The network conditions like congestion, link failure, etc may change during the course of the session and hence data flow from high capacity peers may be low. In the present system there is no mechanism to deal with such a situation. There are two ways in which this problem can be overcome. First, each peer reports its current sending rate periodically to the SuperNode and the SuperNode monitors these rates and makes a decision on swapping.

Another solution will be for a peer to monitor data rates through each of its incoming link and when this rate falls below a threshold it could inform the SuperNode to swap the corrupt peer. The latter solution reduces the burden on the SuperNode by shifting the job of monitoring data speeds to the peers.

3. Action on Link Confirmation: Currently, when links for a peer are updated, the peer sends back a message confirming the new links. The SuperNode then asks all the affected peers to finalize their connection. But if the peer fails to send back a links confirm message then there is no action taken and the SuperNode waits indefinitely. The SuperNode has to receive links confirm from all the affected peers. The links confirm feature was incorporated in the system so that the situation wherein a new link to a peer fails can be handled. The easiest way to handle this situation is to keep a copy of the old topology and then send updates. Only when links are confirmed by all affected peers should the SuperNode change the mesh topology to the new one. If some peer fails to update its link, then the SuperNode could go back to the old topology. This mechanism works well if the mesh was changed in response to a join request. In this case, the SuperNode does not accept the new node. But in case of a leave request or peer failure, this should be handled in another manner.

4. Support for multiple protocols: Currently, the data is transferred using UDP as the transport layer protocol. This is keeping in mind the nature of data that is required to be disseminated by the system. However the system can be extended to support the TCP protocol in which case hard data like text can be disseminated without loss, to multiple recipients.

11. CONCLUSION

The thesis presents a P2P system designed for optimal synchronous real time and non-real time data dissemination from a single source to multiple receivers in a source constrained environment. A source constrained environment refers to one in which nodes have much higher download speeds than upload speeds.

The thesis defines the notion of *Throughput Efficiency* to measure the effectiveness of any data dissemination system in a source constraint network and suggests a topology that tries to achieve maximum throughput efficiency while keeping the delay and out-degree down to acceptable levels. Also, the thesis provides a sample implementation of the system in order to measure the effectiveness of the topology. This thesis looks at various issues involved in node and topology management in designing a real world P2P system using the suggested topology. Finally a set of experiments, including real world deployment of the system provided encouraging results. The proposed system outperformed traditional overlay multicast tree and achieved near optimal throughput. Simulation results were presented to verify among others the robustness of the system in case of node failures. Thus, overall the thesis provides a basic implementation of node management in a P2P system and presents the set of experiment results.

BIBLIOGRAPHY

[1] P.A. Chou, A.E. Mohr, A. Wang, and S. Mehrota, "Error control for receiver-driven layered multicast of audio and video," *IEEE Transactions on Multimedia*, vol. 3, pp. 108–22, March 2001.

[2] W. Tan and A. Zakhor, "Error control for video multicast using hierarchical fec," in *Proceedings of 6th International Conference on Image Processing*, October 1999, vol. 1, pp. 401–405.

[3] P. A. C. Jin Li and C. Zang, "Mutualcast: An efficient mechanism for one-to-many content distribution," *ACM Sigcomm Asia Workshop*, April 2005.

[4] S. Deering et al., "The pim architecture for wide-area multicast routing," *IEEE/ACM Transactions on Networking*, vol. 4, pp. 153–162, April 1996.

[5] S. Banerjee, C. Kommareddy, K. Kar, B. Bhattacharjee, and S. Khuller, "Construction of an efficient overlay multicast infrastructure for real-time applications," in *IEEE INFOCOM*, 2003.

[6] http://www.gnutella.com.

[7] http://www.kazaa.com.

[8] J. Hartigan, *Clustering algorithms*, John Wiley and Sons, Inc, 1975.

[9] Information Sciences Institute, http://www.isi.edu/nsnam/ns, *Network simulator*.

[10] *Internet topology generator*, http://www.cs.bu.edu/brite.

[11] V.N. Padmanabhan, H.J. Wang, P.A. Chou, and K. Sripanidkulchai, "Distributed streaming media content using cooperative networking," in *ACM NOSSDAV*, Miami, FL, May 2002.

[12] T. Nguyen and A. Zakhor, "Multiple sender distributed video streaming," *IEEETransactions on Multimedia and Networking*, vol. 6, no. 2, pp. 315–326, April 2004.

[13] J. Byers, J. Considine, M. Mitzenmacher, and S. Rost, "Informed content delivery across adaptive overlay networks," *IEEE/ACM Transactions on Networking*, vol. 12, no. 5, October 2004.

[14] Y. Minsky, A. Trachtenberg, and R. Zippel, "Set reconciliation with nearly optimal communication complexity," in *IEEE International Symposium on Information Theory*, 2001.

[15] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat, "Bullet: High bandwidth data dissemination using an overlay mesh," in *SOSP*, October 2003.

[16] M. Castro, P. Druschel, A. Kermarrec, A. Nandi, A. Rowstron, and A. Singh, "Splitstream: High-bandwidth multicast in a cooperative environment," in *SOSP*, October 2003.

[17] A. Rowstron, A-M. Kermarrec, M. Castro, and P. Druschel, "Scribe: The design of a large-scale event notification infrastructure," in *NGC*, November 2001.

[18] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," in *IFIP/ACM International Conference on Distributed Systems Platforms*, November 2001.

[19] R. Sherwood, R. Braud, and B. Bhattacharjee, "Slurpie: A cooperative bulk data transfer protocol," in *IEEE INFOCOM*, 2004.

[20] A. Savrou, D. Rubenstein, and S. Sahu, "A lightweight, robust p2p system to handle flash crowds," in *IEEE ICNP*, November 2002.

[21] X. Yang and G. de Veciana, "Service capacity of peer to peer networks," in *IEEE INFOCOM*, 2004.

[22] Z. He, D. Figueiredo, S. Jaiswal, J. Kurose, and D. Towsley, "Modeling peer-to-peer file sharing systems," in *IEEE INFOCOM*, April 2003.

[23] http://www.opencola.org/projects/swarmcast.

[24] http://www.bittorrents.com.

[25] PlanetLab, http://www.planet-lab.org.

[26] Masters Thesis of Rohit Kamath, EECS Department, Oregon State University

[27] Thinh Nguyen, Krishnan Kolazhi, Rohit Kamath, Phuoc Do "Efficient Content Distribution in Source Constriant Networks" in *IEEE Transactions in Mutimedia.*

[28] Thinh Nguyen, Krishnan Kolazhi, Rohit Kamath "Efficient Video Dissemination In Structured Hybrid P2P Networks" in *ICME 2006.*

APPENDICES

APPENDIX A. First Appendix

Proofs: The proofs presented in this section can also be found in [27].

Theorem 1: Throughput efficiency $E \leq 1$ *for any topology and data dissemination algorithm.*

Proof: Throughput efficiency is defined as

$$E \triangleq \frac{\sum_{i=0}^{i=N} S_i}{min(\sum_{i=0}^{i=N} C_i, NC_0)} \tag{A.1}$$

where 0 denotes the source node, $i = 1...N$ denote N destination nodes, S_i and C_i are the *useful* sending rate and the sending capacity of node i, respectively.

Case 1: Assume $\min(\sum_{i=0}^{i=N} C_i, NC_0) = \sum_{i=0}^{i=N} C_i$, then since $C_i \geq S_i$, we have $E = \frac{\sum_{i=0}^{i=N} S_i}{\sum_{i=0}^{i=N} C_i} \leq 1$.

Case 2: Assume $\min(\sum_{i=0}^{i=N} C_i, NC_0) = NC_0$, then $E = \frac{\sum_{i=0}^{i=N} S_i}{NC_0}$. Now, we observe the following. A destination node cannot receive the information at a rate faster than the information rate being injected into the network. Since the source node injects the maximum data rate of C_0 into the topology, maximum total receiving rate of useful data for all N destination nodes is NC_0 bps. The total sending rate $E = \sum_{i=0}^{i=N} S_i$ and the total receiving rate must equal to each other. Therefore the total sending rate is less than or equal to the *maximum* total receiving rate of all the nodes NC_0. Hence, $E = \frac{\sum_{i=0}^{i=N} S_i}{NC_0} \leq 1$.

Theorem 2: For a fully connected topology, the following properties hold.

(a): Throughput efficiency of this scheme $E = 1$.

(b): The maximum node delay D is constant.

(c): Node insertion and deletion for this algorithm can affect at most N nodes

where N is the number of destination nodes.

(d): The out-degree of each node is at most $N-1$ where N is the number of destination nodes.

Proof :

(a): Each destination node receives C/N bps and broadcasts its data to other $N-1$ destination nodes at the rate of C/N each. Hence, a destination node sends packets at a total rate of $(N-1)(C/N)$ bps. Total sending rate from N destination nodes and one source node equals to $(N-1)C + C = NC$ bps. In this scenario, $\min(\sum_{i=0}^{i=N} C_i, NC_0) = NC$, hence $E = 1$.

(b): Each destination node is connected to the source and N - 1 other destination nodes. The maximum number of hops for every data packet to get to a destination node would be 2. Thus, the maximum node delay is constant and makes the scheme optimal in terms of delay.

(c): Whenever a new node joins, it needs to connect to the source and the remaining $N-1$ nodes. This operation will affect $N-1+1 = N$ nodes including the new node itself. Similarly, a leaving node also affects N nodes. The property of the scheme is not optimal in terms of node addition or deletion.

(d) Each node receives data from the source and forwards it to $N-1$ destination nodes that it is connected to. Thus, this scheme is not optimal in terms out-degrees of destination nodes. Note that for the source, the out-degree is N as it is connected to all the destination nodes.

Theorem 3: For a fully chain topology, the following properties hold.

(a): Throughput efficiency of this scheme $E = 1$.

(b): The maximum node delay D is O(N) where N is the number of destination nodes.

(c): Node insertion and deletion for this algorithm affects constant number of nodes.

(d): The out-degree of each node is constant.

Proof :

(a): Each destination node, except the last node in the chain, receives C bps and broadcasts its data to one other destination node at the rate of C. Total sending rate from $N-1$ destination nodes and one source node equals to $(N-1)C + C = NC$ bps. Again $\min(\sum_{i=0}^{i=N} C_i, NC_0) = NC$. Clearly, in this scenario,

$$E = 1 \tag{A.2}$$

(b): Since all the nodes are chained, the maximum number of hops required for a data packet to reach node N from the source would be $N-1+1 = N$. Thus, this scheme performs poorly in terms of node delay.

(c): This scheme involves chaining of nodes. When a new node joins in, it is chained to the last node in the topology and therefore affects only one node. When a node leaves, if it is the last node in the chain, the only node affected is the node that the departing node is chained to i.e. node $N-1$. However, for all

other cases, there will be exactly two nodes affected viz. the node the departing node is chained to and the node that is chained to the departing node. Thus, even when a node leaves, the number of nodes affected is still constant. Hence, this scheme is optimal in terms of node addition and deletion.

(d) Each destination node sends out data to exactly one other destination node and therefore, the out-degree of each node is 1. Note that the out-degree of the last node in the chain is zero as there are no more nodes in the chain to broadcast data.

Theorem 4: For a balanced mesh, the following properties can be proved.
(a): Throughput efficiency of this scheme $E = 1$.
(b): The out-degree for each node is at most b.
(c): The maximum node delay D is $log_b((b-1)N+b)+1$ where N is the number of destination nodes.

Proof :

(a): As shown in the construction algorithm, within a group, there is exactly one rightmost leaf node which does not forward its data to any of its ancestors. This rightmost leaf node, however, forwards its data to $b-1$ leaf nodes at the rate of C/b bps. The rest of the "fully active" nodes within each group forward data at the rate of C bps. Since there are b groups in a b-balanced mesh, the total sending rate of the entire mesh equals to sum of the sending rates of the source node, $N-b$ "fully active" nodes and b rightmost leaf nodes, i.e. $\sum_{i=0}^{i=N} S_i = C + (N-b)C + b(b-1)C/b = NC$ bps. The denominator of E equals

to $\min((N+1)C, NC) = NC$. Hence, the throughput efficiency is $NC/NC = 1$.

(b) : By construction, each internal node has exactly b out-connections to b children. With the exception of the rightmost leaf nodes from each group, each leaf node has $b-1$ out-connections to other leaf nodes, and one out-connection to its ancestor (e.g. parent, grandparent, ...). Thus all nodes have out-degree of b, except the b rightmost leaf nodes from each group which have out-degree of $b-1$.

(c) : Using geometric sum, the total number of destination nodes N and the source node is $N+1 = (b^{(i+1)} - 1)/(b-1)$ where i is the number of levels in the mesh. Hence, hence there are $i = log((b-1)N + b) - 1$ hops from the source node to a leaf node. Next, by construction, there is exactly one hop from a leaf node to another leaf node in a different group. There is also one hop from the leaf node to an internal node. Therefore, the maximum delay for any node is $log((b-1)N + b) + 1$.

Theorem 5: For a cascaded balanced mesh, the following properties can be proved.

(a): Throughput efficiency $E = 1$.

(b): The delay is $O((log_b N)^2)$.

(c): The out-degree for each node is at most b.

Proof:

(a): This holds true since each cascaded mesh is a b-balanced mesh where the root receives data at a rate of C bps. We proved this property for balanced

meshes earlier.

(b) : Our proof relies on the observation that the maximum number of b-balanced meshes of depth i needed to accommodate the remaining nodes at level i is no greater than some constant c. As the algorithm progresses, the new mesh is either equal or smaller than the mesh in the previous iterations, i.e., the depth of the mesh decreases monotonically. Hence, the algorithm terminates after at most some constant c times the depth of first mesh. The constant c indicates the maximum number of meshes of depth i in the cascaded b-balanced mesh. Since there are $O(i)$ such meshes and each mesh has depth of $O(i)$, the total delay is therefore $O(i^2)$, or equivalently $O((log_b N)^2)$. Now, we prove this property precisely. At each iteration of the algorithm, we construct the deepest b-balanced mesh without exceeding the number of nodes. Therefore, the remaining number of nodes after constructing a b-balanced mesh of maximum depth i cannot be greater than b^{i+1}. Otherwise, we can construct a b-balanced mesh of depth $i+1$ which contradicts the maximum possible i. Next, since the number of nodes in a b-balanced mesh of depth i is $(b^{i+1}-1)/(b-1)$, the maximum number of meshes of depth i that can cover the remaining nodes without exceeding the number of possible nodes is therefore $b^{i+1}(b-1)/(b^{i+1}-1) \leq b$. Therefore, we can construct at most b meshes of depth i before moving to the meshes of depth $j < i$. Hence, after the algorithm terminates, we have at most bi meshes with i being the depth of the first mesh. Since each mesh has depth of $O(i)$, the total delay is therefore $O(i^2)$, or equivalently $O((log_b N)^2)$.

(c) : This property is true by construction from the *balanced mesh*

Theorem 6: For a b-Unbalanced mesh with N destination nodes, the following properties can be proved.

(a): Throughput efficiency $E = 1$.

(b): The delay D is at most $\lfloor \log_b(N+1) \rfloor + 3b - 4$.

(c): Node insertion and deletion for this algorithm can affect at most $b^2 + 2b$ nodes.

(d): The out-degree for each node is at most b.

Proof :

(a) : For a b-Unbalanced mesh, when there exists a secondary mesh, it is constructed according to the algorithm for cascaded balanced mesh. We already know that for cascaded balanced mesh $E = 1$. When the secondary mesh is broken, it is reattached to the primary mesh and the primary mesh is reorganized. After all the changes are made, there still remain only b nodes each having C/b unused bandwidth. All the remaining nodes transmit data at C bps. Thus the total unused bandwidth in the system is C. Hence the efficiency in this case too is 1 similar to that in a balanced mesh .

(b) : The delay of the root node in the first secondary mesh is $\lfloor \log_b(N+1) \rfloor + 1$. This is because the root node of the first secondary mesh receives b different partitions from each of b the rightmost leaf nodes in the primary mesh. These partitions take $\lfloor \log_b(N+1) \rfloor$ hops to arrive at the rightmost leaf nodes from the source node, and one more hop to the secondary mesh's root node. Now, the secondary meshes consists of many balanced meshes cascaded together as described in Section 3.4. Each of these balanced mesh has at most one level since a balanced mesh with two levels would result in the number of nodes equals to

$(b^3-1)/(b-1) > b^2-1$, which is not possible by design. The largest delay then occurs when the number of nodes in the secondary meshes is b^2-2 since in that case, the secondary meshes must consist of $b-2$ balanced meshes, each mesh with $b+1$ nodes, followed by a chain of b nodes. Since there are two hops from the root of one balanced mesh to the other and $b-1$ hops connecting the chain of b nodes, the largest delay equals to $2(b-2)+b-1 = 3b-5$ hops. We sum this delay and the delay to the root of the first secondary mesh. We note that if the out-degree o_i of each balanced mesh in the secondary meshes is changed adaptively (o_i can be less than b to satisfy the constraint on out-degree), the overall delay for the small mesh can be smaller than $3b-5$ hops and the throughput efficiency still equals to 1.

(c) : The most number of nodes are affected when there is a construction or destruction of secondary meshes. In this case, at most b^2 nodes belonging to the secondary meshes are affected. In addition, there are b nodes that these b^2 nodes are attached or detached during the construction or destruction of the secondary meshes. Furthermore, there are also b ancestors, one from each branch that need to receive data from the new b nodes (e.g. node 3 in Figure 3.5(b)).

(d) : By construction, each internal node has exactly b out-connections to b children. With the exception of the rightmost leaf nodes from each group, each leaf node has $b-1$ out-connections to other leaf nodes and one out-connection to its ancestor (e.g. parent, grandparent, ...). Thus, all nodes have out-degree of b, except the b rightmost leaf nodes from each group which have out-degree of $b-1$. In case of the chained nodes, they have only one outgoing connection and for the secondary root with no children, it has no outgoing connections.

APPENDIX B. Second Appendix

Algorithm APPENDIX B.1: BalancedMesh(N)

Construct balanced tree with source as the root and each internal node
with out degree of b.

comment: Consider the leftmost group as group 0 and the rightmost as group b - 1.

comment: Assume there are *depth* levels in the tree.

for i←0 **to** b - 2

do
- **for each** Leaf node j in group i
  - **do**
    - **for** m←1 **to** (b - 1) - i
      - **do**
        - $k \leftarrow j + b^{depth-1}m$
        - Connect node j with node k.
        - Connect node k with node j.

for i←0 **to** b - 1

do
- Connect leftmost b - 1 leaf nodes of group i back to its parent.
- Connect the rightmost leaf node of each branch to its lowest ancestor without b incoming connections.

Algorithm APPENDIX B.2: CascadedMesh(N)

while $N <> 0$

do

Construct a b balanced mesh of depth

$i \leftarrow \lfloor log((b-1)N + b) \rfloor - 1$.

comment: The above statement will create the deepest b balanced mesh.

comment: Number of nodes in the mesh should not exceed the N.

if exists previous b balanced mesh

then Connect b rightmost nodes with extra bandwidth to the root of newly created balanced mesh.

$N \leftarrow N - (b^{i+1} - 1)/(b-1)$.

Algorithm APPENDIX B.3: unBalancedMesh(i)

if $sec\_mesh\_node\_count == 0$

then
- Set it as root of the secondary mesh.
- $sec\_mesh\_node\_count \leftarrow sec\_mesh\_node\_count + 1$.
- **return** (0)

if $sec\_mesh\_node\_count < b^2 - 1$

then
- Add the node using b cascaded balanced mesh algortihm.
- $sec\_mesh\_node\_count \leftarrow sec\_mesh\_node\_count + 1$.
- **return** (0)

comment: Do following if added node i causes b^2 nodes in secondary mesh.

comment: In this case the secondary mesh is destroyed.

comment: Let the leftmost group be 0 and the rightmost be b - 1.

for i←0 **to** b - 1

do
- **if** All leaf nodes of primary mesh are at same level
  - **then** Connect b nodes of secondary mesh to leftmost node P in group i.
  - **else**
    - Connect b nodes of the secondary mesh to leftmost leaf node P
    - of lesser depth in group i
- Disconnect Ps connection to b - 1 nodes of other b - 1 groups.
- Disconnect Ps connection back to its parent.

for each Group of newly attached b nodes

do
- Establish their cross links with other groups as described for balanced mesh.
- Connect all nodes but the rightmost node to their parent P.
- Connect rightmost node to lowest ancestor without b incoming connections.

$sec\_mesh\_node\_count \leftarrow sec\_mesh\_node\_count + 1$.

return (0)

Algorithm APPENDIX B.4: LEAVENODE(i)

if node is in primary mesh

then

- **if** secondary mesh exists
  - **then**
    - Swap leaving node with node in secondary mesh.
    - Reconstruct the secondary mesh.
- **else if** node is internal node
  - **then**
    - Swap the node with a leaf node in the primary tree.
    - Construct a secondary mesh with $b^2 - 1$ nodes.
- **else**
  - **comment:** No need of swapping if node is not internal node.
  - Construct a secondary mesh with $b^2 - 1$ nodes.

else

comment: Node is secondary mesh.

- Reconstruct the secondary mesh with one lesser node.