AN ABSTRACT OF THE THESIS OF

Rohit Kamath for the degree of Master of Science in Computer Science presented on June 1, 2006.

Title: Efficient Content Distribution on Source Constraint Networks: Peer Communication

Abstract approved: _____

Thinh Nguyen

Popular applications such as P2P file sharing, multiplayer gaming, video-conferencing, etc. rely on the efficiency of content distribution from a single source to multiple receivers. Most users of these applications are on the widely prevalent source constraint networks such as the Digital Subscriber Line (DSL) and wireless networks. Overlay multicast techniques are widely employed to address this issue. However, most approaches do not consider the source constraint nature of the underlying network nor do they achieve optimal system throughput in the overlay structure. We propose various topologies in which all participating nodes contribute their bandwidth, resulting in high overall system throughput. We implement *Hypp*, a hybrid P2P system, based on the proposed topology. *Hypp* consists of one or more *Supernodes* that perform mesh and node management functions, and participating *Peers*, connected together in a structured manner as directed by a Supernode. This work focuses on the *Peer* subsystem that is responsible for communication and actual content delivery. We present results of experiments conducted over PlanetLab nodes that prove *Hypp* is a scalable and high throughput system capable of synchronous and efficient content delivery.

Efficient Content Distribution on Source Constraint Networks: Peer Communication

by

Rohit Kamath

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented June 1, 2006
Commencement June 2007

Master of Science thesis of Rohit Kamath presented on June 1, 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.

Rohit Kamath, Author

ACKNOWLEDGMENTS

I would like to thank God for making everything possible. I would like to express my deepest gratitude to my academic and research advisor, Dr. Thinh Nguyen, for his encouragement, motivation, guidance and support in helping me to complete this work. I would also like to express my gratitude to my teammates, Krishnan Kolazhi and Phuoc Do, for their contributions, ideas and opinions towards making this possible.

I would like to thank Dr. Michael Quinn, Dr. Timothy Budd and Dr. Rajeev Pandey for their insights during courses that assisted me in my research and resulted in a lot of knowledge gain. I would like to thank Dr. Bose for all his help.

I would like to thank Sunand Tullimalli, Ankit Khare, Ben Hermens, Roshan Urval, Santosh Tiwari, Anand Venkataraman, Vaidyanathan Ramamoorthy and Nilesh Araligidad for being such great friends during my term as a graduate student at Oregon State University.

I wish to thank my fiancee, Pradnya, my friends, Ram, Ashay, Zubin, Nikhil, Asim, Maulin and my family for their love, affection, encouragement and unconditional support all along.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

Page

TABLE OF CONTENTS (Continued)

Page

TABLE OF CONTENTS (Continued)

Page

LIST OF FIGURES

Figure Page

LIST OF FIGURES (Continued)

Figure Page

Figure Page

LIST OF TABLES

Table Page

Efficient Content Distribution on Source Constraint Networks: Peer Communication

1. INTRODUCTION

1.1. Problem Domain

Improvement in the network delivery infrastructure and powerful end system computational resources have led to the steady development of new and exciting networked applications that have the potential to revolutionize our lives. Prime examples of such applications are P2P file sharing, multi-player gaming, video conferencing, distance learning, remote presentations, live video streaming applications and their kind. The vast potential of such applications can be described by citing a few simple examples. Imagine being able to attend a lecture from a remote location just by "tuning in" and being able to see and hear the professor deliver his content. A lot of sites offer packages for watching various sporting events live. Another instance that can be cited is that of an end user being able to make a presentation to clients who are spread far and wide across the globe.

All these exciting prospects basically deal with the issue of efficient and synchronous content distribution from a single source to multiple receivers. However, the underlying networks for most of these application users are the widely prevalent *source constraint* networks. 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. A node

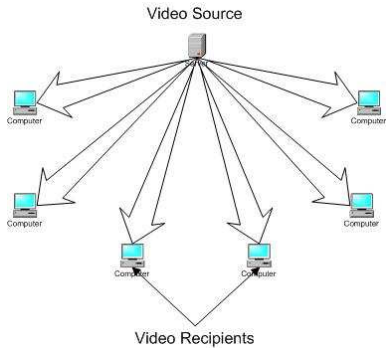


FIGURE 1.1. *Content delivery from a single source to multiple receivers*

may allocate its sending rates to each of its neighbors as long as the total sending rate does not exceed the capacity. In other words, a node can have multiple links to other nodes, but the total bandwidth is constrained to a maximum value. Thus, hosts on source constraint networks possess high download bandwidths but relatively very low upload capacities. Examples of such networks can be cited as the Digital Subscriber Line (DSL) networks and wireless networks. If the source and the content requesting nodes are on such source constraint networks, then achieving efficient content distribution from the source to the receivers is a challenging task that we seek to address.

1.2. Background

As observed in the Figure 1.1, a significant factor that governs the performance of these applications is the efficient content delivery from the source to all the destination nodes scattered randomly across the globe. A few traditional techniques that were tried out are discussed below.

1.2.1. Replicated Unicast

Most high-level network protocols (TCP or UDP) only provide a unicast transmission service and, therefore, nodes have the ability to send to only one other node at a time. All transmission with a unicast service is inherently point-to-point. As shown in Figure 1.2, if a node wants to send the same information to many destinations using a unicast transport service, it must perform a replicated unicast, and send N copies of the data to each destination in turn. This technique is highly inefficient for reasons such as the unnecessary overhead of generating multiple copies of the data at the source and the inability to keep up the transfer speed at a desired bitrate to achieve efficient and smooth content delivery in case of streaming media.

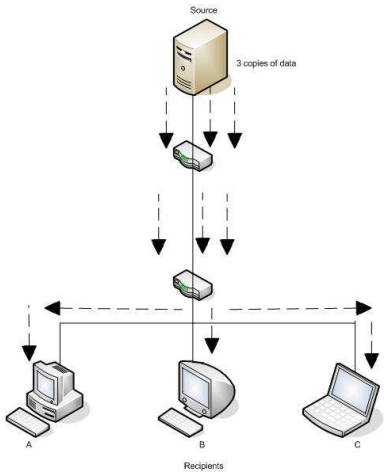


FIGURE 1.2. *Replicated Unicasting*

1.2.2. IP Multicast

The serious drawbacks of replicated unicasting led to the emergence of multicasting techniques such as the traditional IP Multicast. IP multicast works at the network layer and all the complex functionality of data delivery is pushed out into the routers which essentially got rid of the problem of the source having to generate multiple copies of the same data. Additional information was provided to the routers that enabled them to deliver data to reachable destination nodes if they detected that the destination nodes were subscribers of the multicast group that the data was addressed to. Refer to Fig 1.3. However, the drawback of this approach was that it placed a lot of burden on the routers that were already stressed out executing their basic packet switching functions. Also, the topology employed did not allow for optimal system throughput and there was considerable room for improvement. There also were compatibility issues across Autonomous Systems (AS) where not all routers supported multicasting.

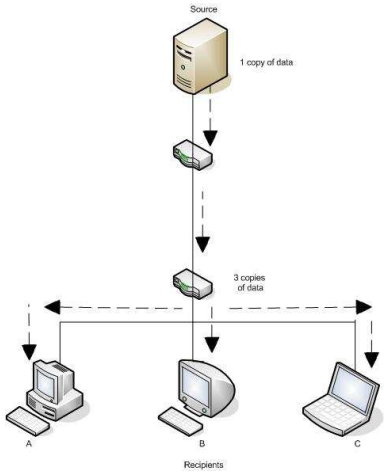


FIGURE 1.3. *IP Multicast*

1.2.3. Overlay Multicast

The next logical approach was the *Overlay Multicast* (also known as End-System multicast or Peer-to-Peer Multicast). In this approach, all the complex functionality such as node organization and data delivery was delegated to the application layer in powerful end systems thereby relieving the underlying routers of the multicasting overhead. This also allowed for deployment across different AS(es) without any issues. Hence, most current streaming applications employ such overlay network models to disseminate data from a source to multiple recipients. However, most overlay multicasts do not deliver optimal performance as they do not take the source constraint nature of the underlying networks into consideration. Moreover, identical packets may travel on the same physical links due to the inability of the application layer to control the underlying routing. Leaf nodes in the overlay structures do not contribute their resources to the overall system throughput. Another factor for sub-optimal overlay multicast performance is the employment of inefficient data partitioning techniques. Two widely used overlay network topologies are the *tree* and the *mesh* structures. Consider an overlay *tree* topology in Figure 1.4(a) in which the source wants to disseminate a large file to all the overlay nodes. If the sending rate of the source to destination 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. Thus, an overlay *tree* is highly inefficient.

Let us now consider a different overlay *mesh* topology in Figure 1.4(b) where there are additional links between the destination nodes. In particular, consider the link from C to B. Assume further that node C has bandwidth of

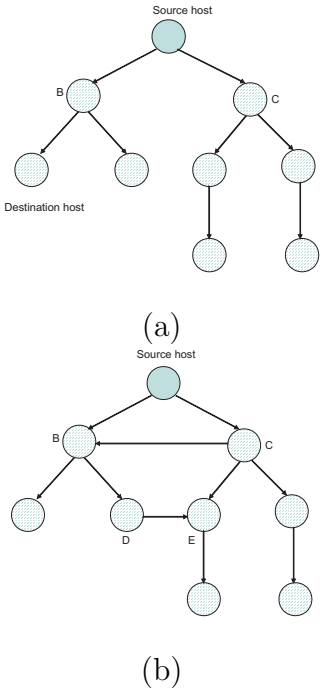


FIGURE 1.4. *Examples of (a) an overlay multicast tree; (b) a mesh topology*

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 resulting in a bandwidth improvement of four times over the overlay multicast tree approach. Thus, employment of proper data partitioning techniques and an efficient topology can lead to improvements in performance. One of many such inefficient overlay multicasts is the *Vanilla Multicast* as depicted in Figure 1.5.

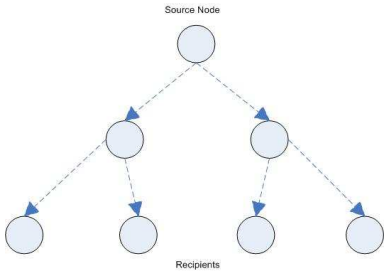


FIGURE 1.5. *Vanilla Multicast*

1.3. Keywords

Overlay Multicast, P2P, Bandwidth Asymmetric Networks, Content Distribution, Topology, Throughput efficiency, Data Dissemination.

1.4. Scope and Objectives

We propose an overlay multicast technique that achieves efficient, scalable and synchronous content delivery from a single source to multiple receivers in bandwidth asymmetric networks. To achieve our objectives, we focus on the following factors :

1. Topology of the overlay network, and
2. Contributions from each participating node towards increasing 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 source constraint networks such as DSL and wireless networks.

2. We also assume that the upload capacities of all nodes are approximately the same in order to simplify our discussion. However, later on, we present an optimization technique that minimizes the performance degradation in case of nodes with unequal upload capacities.

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 the complete data without damaging the connectivity of the remaining nodes.

3. End-to-end delay from the source node 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 the nodes is optimal in the sense of average useful throughput, a quantity defined in Section 3.1.

   We intend to address these issues by :

   (a) defining the notion of throughput efficiency.

   (b) presenting an algorithm to design an application layer mesh topology that exhibits near optimal throughput by having all nodes, including the leaf nodes, contribute to the overall system throughput.

(c) designing and developing a system based on the proposed topology.

(d) discussing the design of the peer subsystem in detail.

(e) evaluating the performance of the system based on certain key factors which are discussed later.

1.5. Overview

The rest of this work is organized as follows. Chapter 2 describes some of the related work that has been carried out in the problem domain and the motivation behind our approach. We commence addressing the issue by defining the notion of throughput efficiency in Chapter 3 to measure the performance of any topology and data dissemination algorithm. 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. In Chapter 4, we present a high level design of an implemented hybrid P2P system, *Hypp*, based on the proposed topology. Chapter 5 discusses some of the common subsystems of the hybrid P2P system, *Hypp*. Chapter 6 discusses the *Peer* subsystem of *Hypp* in detail. Chapter 7 presents and analyzes the results of experiments conducted over PlanetLab [29] nodes. We also discuss the results of some large scale simulations run using NS [12]. Chapter 8 contains our conclusions and discusses some of the scheduled future work.

2. LITERATURE REVIEW

In the following section, we list related work on data dissemination on the Internet.

2.1. Related Work

Related work has been carried out in MutualCast [2] where upload bandwidth of all the nodes is fully utilized. However, the topology employed in MutualCast is essentially a fully connected topology with no constraints on the out-degree of a node. A MutualCast network also consists of non content requesting nodes that simply contribute their resources to the system. Thus, affected nodes during node insertion or deletion would also be $O(N)$ where N is the *total* number of nodes in the MutualCast network. Similar to our approach, Byers et.al. [17] use data partitioning techniques and have peers contribute towards increasing the throughput of the system. In this approach, each node randomly sends different partitions onto different links and data reconciliation techniques [18] are required to reduce the data redundancy between nodes. To address the transient and asynchrony issues of nodes joining and leaving the network, the paper advocates Forward Error Correction (FEC) approach in which a node can successfully recover the entire file using a fraction of the received packets. Similar work has also been done in [19] where the goal is to distribute data to a set of nodes in an overlay multicast mesh resulting in disjoint data sets at each of these nodes. The nodes then can establish concurrent connections amongst themselves in order to increase the download rates. Data reconciliation techniques similar to [17] are used to reduce overlapped data. Both of the above mentioned works focus on protocols and techniques for dynamically exchanging information between the

nodes. On the other hand, this work focuses on constructing a topology with the emphasis on throughput efficiency, node out-degree, node delay and bandwidth fairness. Moreover, unlike the randomized data partitioning techniques employed by others, our data partitioning algorithm is simple, deterministic and results in a small number of partitions thereby requiring no data reconciliation. CoopNet [14] uses multiple overlay multicast trees to stream multiple descriptions of the video to the recipients. Each multicast tree contains a description of the video and the receipt of a large number of descriptions results in a higher quality of video. Thus, the focus of this paper is on reliability and video quality. Most similar to our work is SplitStream [20] in which multiple multicast trees are constructed such that an internal node of one tree has to be the leaf node in the others to improve reliability. Data partitioning is performed and disparate partitions are sent onto different multicast trees. Unlike our work, SplitStream relies on Scribe [21] and Pastry [22] infrastructure for tree construction without any regard for the constraints on out-degree and capacity of each node. P2P networks such as Gnutella [6], KaZaA [7], Swarmcast [27], and BitTorrents [28] are also related to our work in that they allow a node to download contents simultaneously from multiple peers. Similar to our approach, in BitTorrents, a peer can transmit its partially downloaded contents to peers who need them while concurrently downloading other parts of the file from peers that have those parts. Other similar works include [23] which proposes a protocol for cooperative bulk data transfer. Many other related works also propose to offload a server's bandwidth to peers when the number of destination nodes is large, resulting in a highly bandwidth scalable network. For example, authors in [24] 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 [25] [26].

2.2. Motivation

We note that most of the approaches address issues such as reliability and quality. Some of the approaches are not synchronous and therefore, not suitable for applications that require live streaming media delivery. Also, some others do not achieve optimal system throughput due to all nodes not contributing their resources towards the overall system throughput. Only a few existing approaches take into account the source constraint nature of the underlying networks where the upload capacity of a node is the bottleneck. Taking into account these shortcomings, we believe a better approach is possible. We believe our proposed techniques achieve near optimal system throughput while considering all of the factors mentioned above and in addition, achieving a satisfactory tradeoff between node delay and out-degree while maintaining near optimal throughput efficiency.

3. THROUGHPUT EFFICIENCY AND PROPOSED TOPOLOGY

Work similar to this section has been carried out in [31] [32]. In this section, we propose topologies that maximize the throughput efficiency and at the time, achieve a reasonable trade-off in delay and out-degree of a node. However, to measure the performance of different data dissemination schemes in source constraint networks, we define throughput efficiency in the following section.

3.1. 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 node 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 *useful* sending rate S_i is the total rate at which the data is sent directly from node i to all its neighboring nodes such that this data is completely disjoint with the data received at each of its neighboring nodes from all other nodes $k \neq i$. Clearly, the *useful* sending rate S_i is governed by the following factors :

1. Topology, and
2. The algorithms for data dissemination

A data dissemination scheme (which includes both topology and data dissemination algorithm) is considered not optimal if it results in duplicate data partitions at a particular node. Thus, node i does not send a data partition to

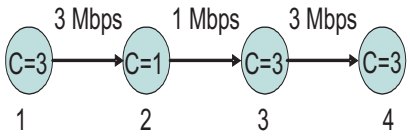


FIGURE 3.1. *Chain topology with throughput efficiency of 0.5.*

a neighbor if that neighbor receives the identical data partition from some other node. As a result, the actual sending rate would equal the useful sending rate of a node. The numerator in the Definition 1 is the total actual sending rate of all the nodes, while the denominator is the minimum of the two quantities: (a) total maximum sending capacity of all the nodes and (b) the maximum receiving capacity. As described, the useful sending rate depends on the topology and data dissemination algorithms at each node. For example, Figure 3.1 shows a chain topology with four nodes and their upload capacities C_i. To disseminate data to all the nodes, node 1 sends packets at its capacity of 3 Mbps. Node 2 receives data from node 1 and relays the packets to node 3. However, it can only send packets at its capacity of 1 Mbps. As a result, node 3 sends packets at 1 Mbps since it receives only 1 Mbps from node 2, even though its capacity is 3 Mbps. The throughput efficiency in this scenario, is therefore $\frac{3+1+1}{3*3} \sim= 0.55$.

Now, if the node 2 is moved to the last position in the chain, it is obvious that the throughput efficiency is now $\frac{3+3+3}{3*3} = 1$. Clearly, to minimize the time to disseminate the data, the topology that results in higher efficiency is preferred. The following proposition helps us to determine the performance bound for any topology and data dissemination algorithm.

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

We prove the above proposition in Appendix A.

Clearly, throughput efficiency relates directly to the average receiving throughput of all the nodes. Efficiency of 1 implies all nodes are sending data at their capacities, and therefore results in highest efficiency. Since the total sending capacity of all the nodes may be larger than the allowable receiving rate as this rate is dictated by the injected data rate by the source, the *min* term in the denominator ensures that the efficiency is not reduced for a network topology with large capacity but a small injected data rate.

3.2. Fully Connected Topology

With the above definition of throughput efficiency, we now discuss the case where the source and all N destination nodes have identical upload bandwidth C bps. Furthermore, no out-degree constraint is imposed on each node. In this special case, we can construct the fully connected topology as shown in Figure 3.2 in which, any node i is connected to $N-1$ other destination nodes. The data partitioning algorithm is as follows. The source divides data into N partitions and sends each partition to N destination nodes at an equal rate of C/N bps. Each destination node then broadcasts the data it receives to $N-1$ destination nodes at a rate of C/N bps per destination node.

Theorem 2 : For a fully connected topology, the following holds true.

1. The throughput efficiency for this scheme $E = 1$.
2. Node delay is constant.

3. Node insertion and deletion affects at most $O(N)$ nodes where N is the number of destination nodes.

4. The out-degree of any node in this scheme is $O(N)$ where N is the number of destination nodes.

We prove the above in Appendix A. The above properties of the scheme suggest that it performs poorly with regards to nodes affected during insertion and deletion operations. Also, there is no constraint on the out-degree of a node.

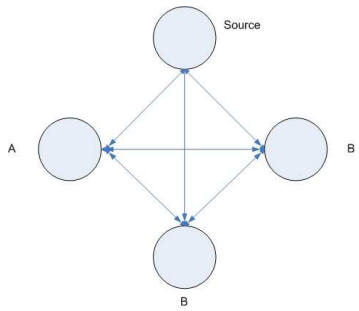


FIGURE 3.2. *Fully Connected Topology*

3.3. Chain Topology

We now discuss the case where we have no constraint on the delay, but we want to minimize the out-degree for each node. Similar to the previous case, we assume that all nodes have the same upload bandwidth C bps. As shown in Figure 3.3, a topology for minimizing node out-degree is a single chain of nodes with the head being the source. Every node in the chain relays data from the source to its neighbor in the chain at the rate C bps.

Theorem 3 : For a chain topology, the following properties are satisfied.

1. Throughput efficiency $E = 1$ for this scheme.

2. Node delay is $O(N)$ where N is the number of destination nodes.

3. Node insertion and deletion affects a constant number of nodes q.

4. Out-degree of a node is constant.

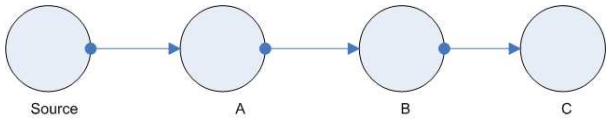


FIGURE 3.3. *Chain Topology*

We prove the above properties in Appendix A. As observed, a significant drawback of this scheme is the extremely large node delay.

3.4. Balanced Mesh

Previous cases are the extreme examples for minimizing delay or node out-degree. In this section, we suggest a topology that achieves a reasonable tradeoff between the delay and node out-degree. In this approach, we assume again that all the nodes have the same upload bandwidth C bps. We build a data dissemination topology that results in a large throughput efficiency, small delay and out-degree of each node no more than b. We construct such a topology as a balanced mesh. A balanced mesh is first constructed as a balanced tree with the source as the root. The leaf nodes of the tree are then connected together and also connected to their ancestors in a systematic manner to result in an efficient data dissemination mesh topology. Figure 3.4 shows an example of a balanced mesh topology with $b = 2$.

In this example, the source partitions the data into two distinct groups and distributes them to the left and right branches of the mesh. Thus, all the leaf nodes under the same branch receive identical data partitions. In order for the leaf nodes to receive both data partitions, each leaf node connects to a leaf node from the other group. For example, Figure 3.4 shows pairs of nodes 7 and 11, 9 and 12, 9 and 13, 10 and 14 connected together. As a result, all the leaf nodes now receive the complete set of data partitions. However, the internal nodes, e.g. nodes 3 and 4 in the left branch are yet to receive the data partition from the right branch. As constructed so far, each leaf node currently sends useful data at only half of its capacity since it is connected to only one other leaf node. To fully utilize its bandwidth, each leaf node first forwards the data partition received from the other group to its parent. If its parent already receives the identical data partition from its other sibling, the leaf node forwards the data partition to its grandparent. The process continues until all the nodes in the mesh receive the complete set of data partitions. For example, node 7 forwards data partition from the right group to its parent (node 3). Since node 3 already receives that data partition from node 7, node 8 forwards its data partition from the right group to its grandparent (node 1). Node 9 forwards the data partition to its parent (node 4) while node 10 does not forward any data partition to any ancestor since there is no node in need of any data partitions in its group. The nodes in the other group also behave in a similar manner. Thus, all nodes receive all data partitions and can reconstruct the complete data.

We now present the general algorithm for constructing a b-balanced mesh. The algorithm ensures that :

1. All the nodes in the balanced mesh receive the complete set of data partitions,

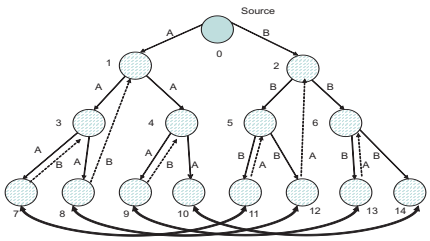


FIGURE 3.4. *Illustration of balanced mesh construction.*

2. No node has out-degree of more than b,

3. The number of hops from the source to the destination nodes is $O(\log_b N)$ where N is total number of nodes, and

4. Throughput efficiency $E = 1$.

To describe the algorithm, we first label the nodes as shown in Figure 3.4. In particular, nodes are labeled from low to high in a breadth-first manner. Within a level, the node labels increase from left to right. The algorithm for constructing the balanced mesh is as follows.

1. Construct a balanced tree with each internal node having out-degree of b with the source being the root.

2. 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 remaining rightmost $b-1$ groups. In particular, node j in the leftmost group g is connected to nodes $k = j + b^{i-1}m$ where $m = 1, 2...b-1-g$. This process is continued for all the groups g where $g = 0, 1...b-2$. Note that, these connections are bi-directional.

3. Each leaf node of the same parent from left to right except the rightmost node, is connected back to its parent. The rightmost leaf node is then connected to the grandparent. Using this construction, each parent node will have b incoming connections; 1 connection from its parent and $b-1$ connections from $b-1$ of its children. The grandparent will also have b incoming connections, 1 connection from its parent and $b-1$ connections from $b-1$ of its grandchildren. The reason for this is that there is exactly one grandchild from each of $b-1$ parents that connects to the grandparent. Next, the grandchild of the rightmost parent is then connected to its grand-grand parent. The process continues until all the internal nodes or ancestors have exactly b incoming connections. Note that by construction, there will be exactly one rightmost leaf node within each group, e.g. nodes 10 and 14 in Figure 3.4, that will not connect to any ancestor.

The pseudocode for constructing a balanced mesh topology can be found in Appendix B.

Given the balanced mesh, the data dissemination algorithm is as follows.

1. The source partitions the data into b distinct groups and sends them down onto b branches of the mesh at the rate of C/b bps per branch. Each internal node in turn broadcasts the data down to its children also at the rate of C/b bps per link.

2. Since each leaf node is connected to $b-1$ other leaf nodes in other groups, a leaf node can forward its data to $b-1$ other leaf nodes in other groups at the rate of C/b bps. As a result, each leaf node receive the complete data from $b-1$ different leaf nodes and its parent.

3. By construction, a parent is connected to $b-1$ children, and therefore, $b-1$ children forward $b-1$ different partitions to their parents at the rate of C/b bps. This is possible because all the children, i.e. leaf nodes have the complete set of data partitions. As a result, a parent node is able to receive the complete data. Similarly, all the ancestor nodes receive the complete data since they all have b incoming connections from the leaf nodes with each leaf node delivering a different data partition.

We now proceed to mention certain properties about the balanced mesh.

Theorem 4 : For a balanced mesh topology, the following holds true.

1. The throughput efficiency $E = 1$.
2. The maximum node delay D is $log_b((b-1)N+b)+1$ where N is the number of destination nodes.
3. The out-degree of any node is at most b.

We prove the above in Appendix A.

3.5. Cascaded Balanced Mesh

In the previous scenario, the total number of nodes must be of the form $(b^i-1)/(b-1)$ where $i, b \in 0, 1, ...$ This is rather a restricted scenario. We now show an algorithm for constructing a mesh with arbitrary number of nodes, and still preserves the desired properties, namely high throughput efficiency, low delay and small out-degree. The main idea of the algorithm is to cascade a series of the balanced meshes in order to accommodate arbitrary number of nodes.

The crucial observation in devising the new algorithm is that the rightmost leaf node of each group has only $b-1$ out-connections, which result in a total of C

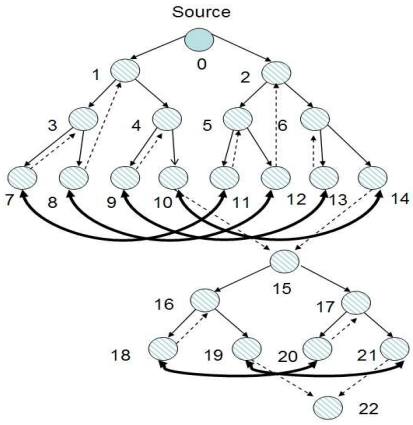


FIGURE 3.5. *A cascaded 2-balanced mesh.*

bps unused capacity. Hence, we can construct a new balanced mesh with the new root connected to b rightmost leaf nodes of the previous balanced mesh. Useful data is then sent from these nodes to the new root at the rate of C/b bps each, or a total rate of C bps. The new root then disseminates data to all the destination nodes in the same manner as described in the case of the balanced mesh. The number of balanced meshes depends on the number of nodes in the mesh. Figure 3.5 shows an example of cascaded 2-balanced mesh consisting of 23 nodes. As seen, the remaining two nodes 10 and 14 of the previous balanced mesh have spare capacity to send their data to the new root node. Similarly, the nodes 19 and 21 send the data to the final node 22, the root of a new mesh without any children.

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

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 constructing a cascaded mesh topology can be found 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 topology,

1. The throughput efficiency $E = 1$.

2. The delay for the cascaded b-balanced mesh is $O((log_b N)^2)$.

3. the out-degree of any node is at most b.

We prove the above in Appendix A.

3.6. b-Unbalanced Mesh

Some drawbacks of the cascaded balanced mesh topology are

1. Node delay is large, i.e. $O(\log_b N)^2$, and

2. Departing nodes cause a large portion of the mesh to be rebuilt.

Therefore, in this section, we introduce a construction that achieves the desired tradeoff between node delay and the number of affected nodes due to other nodes leaving or joining the topology.

Similar to Section 3.5, cascaded balanced meshes are used to accommodate the new nodes. For convenience, we denote the mesh containing the source node as *primary* mesh and the other meshes connected to the primary mesh as *secondary* meshes. We achieve low delay by limiting the number of secondary meshes to a small value. This is done by restricting the total number of nodes in the secondary meshes to $b^2 - 1$. This restriction allows for quick reconstruction of meshes to accommodate nodes entering and leaving the topology. When the number of nodes in the secondary meshes equals to b^2, the secondary meshes are destroyed and their nodes are then attached appropriate places in the primary mesh to achieve the desired throughput efficiency and low delay. The reason behind limiting the number of nodes in the secondary meshes to b^2 is that this is the smallest number of nodes that can be attached at the right places in the primary mesh to maintain the throughput efficiency of 1.

3.6.1. Maintaining the b-Unbalanced Mesh when new nodes join

Assume that we already have a balanced *primary mesh* in place. When a new node joins the mesh, we designate it as the root node of a *secondary mesh*. Now, when another node joins in, it is added to the secondary mesh using the algorithm for constructing the b-balanced mesh as described in Section 3.4. This process is employed for all new nodes and continues until the number of nodes in the secondary meshes is fewer than b^2. Once the node count in the secondary

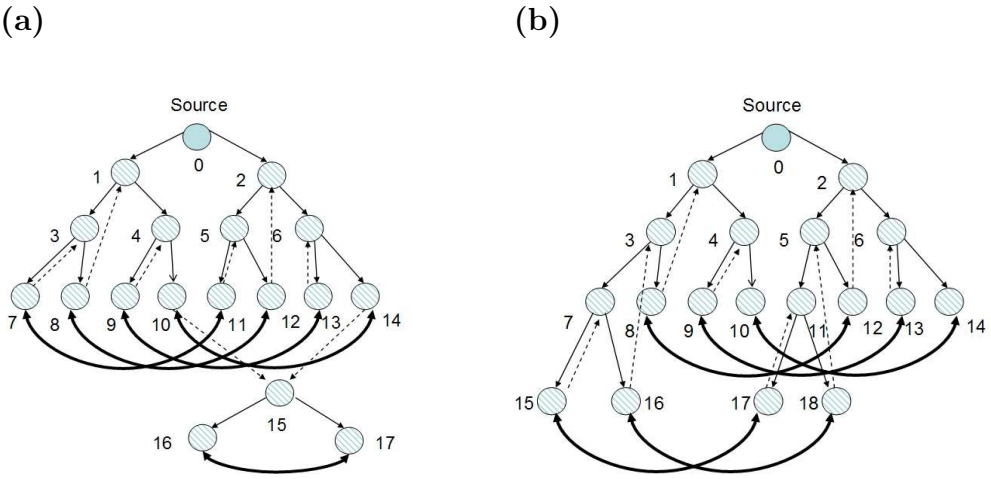


FIGURE 3.6. *(a) The mesh immediately before the destruction of the small mesh; (b) immediately after the destruction of the small mesh.*

meshes reaches b^2, the *secondary mesh(es)* are destroyed and their nodes are attached to the *primary mesh* as described below.

1. If the depth i of all leaf nodes is equal, the first b new nodes will be attached to the first node from left to right within the first group. The next set of b new nodes will be attached to the first node from left to right within the next group and so on.

2. Suppose the leaf nodes of the primary mesh differ in depth by 1 as a result of *case 1* being executed previously. In particular, consider that the leaf nodes have depths of i and $i + 1$. In this case, the first b secondary mesh nodes will be attached to the first node of depth i from left to right within the first group. The next set of b new nodes will be attached to the first node of depth i from left to right within the next group and so on.

This process continues until the last b new nodes are attached to the appropriate node in the rightmost group as described above.

When the b new nodes are attached to a node P in the primary mesh, node P disconnects

1. $b-1$ connections to nodes in the other $b-1$ groups, and
2. one connection that is used to forward data from another group to its ancestor.

With the availability of the released b connections, node P can relay data from its parents to the b newly attached secondary mesh nodes. The b newly attached nodes are then connected to the newly attached nodes in other groups in a similar manner as described in the balanced mesh topology. Since the ancestor of node P no longer receives data from the other group, the rightmost node of the new b children of P delivers data to P's ancestor. Now, if more nodes join in, a new secondary mesh is constructed. After b^{j+1} new nodes join, where j is the depth in the primary mesh, the primary mesh is balanced and its depth increases by 1. Figure 3.6 illustrates the incremental construction of a 2-unbalanced mesh. Initially, the primary mesh consists of 15 nodes. Figure 3.6(a)shows the resultant topology after 3 new nodes join. When a new node joins, the number of nodes in the secondary mesh reaches b^2. As a result, the secondary mesh is destroyed and its nodes are attached to the nodes in the primary mesh. Nodes 15 and 17 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 relay data from the other group to their ancestors. Instead, these two extra connections are used to deliver data to the new nodes. Nodes 15 and 16 then exchange data with nodes 17 and 18. Node 15 also delivers data from the other group to its parent (node 7) and node 16

forwards data from the other group to its grandparent (node 3). Nodes 17 and 18 also behave in a similar manner. Thus, all the nodes in the topology receive complete data.

The pseudocode for adding a node to an unbalanced mesh can be found in Appendix B.

3.6.2. Maintaining the b-Unbalanced Mesh when nodes leave

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

1. If there exists a secondary mesh, select a node from the secondary mesh to replace the departing node. This replacement ensures that the structure of the primary mesh is unaffected. Next, rebuild the secondary mesh(es).

2. If there is no secondary mesh and the departing node is not of the largest depth, select a leaf node from the primary mesh with the largest depth to replace the departing node. Next, construct a secondary mesh consisting of the $b^2 - 1$ nodes. The $b^2 - 1$ nodes are

   (a) the siblings of the replacement node, and

   (b) the nodes in other groups that connect directly to the replacement node and its siblings.

3. If the departing node is of the largest depth, node replacement is not necessary and a secondary mesh consisting $b^2 - 1$ nodes associated with the departing node is constructed. The $b^2 - 1$ nodes are selected as described above.

4. If the departing node belongs to a secondary mesh, rebuild the secondary mesh.

It can be proved that in the unbalanced mesh topology, the number of nodes affected by a node removal or insertion is at most O(b^2) and the delay is of O($log_b N$). 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 handling a node departure in an unbalanced mesh topology can be found in Appendix B.

We enumerate the important properties of this technique as follows.

Theorem 6 : For an b-unbalanced mesh topology, the following properties hold:

1. Throughput efficiency for this scheme $E = 1$.
2. Node delay is $O(\log_b N) + c$ where c is a constant.
3. Node insertion and deletion for this algorithm can affect at most $b^2 + 2b$ nodes.
4. Out-degree of any node is at most b.

We prove all of the above in Appendix A.

4. SYSTEM ARCHITECTURE

Hypp is our hybrid P2P experimental system based on the proposed topology. Similar work has been done in [31] [32]. *Hypp* stands for a **Hybrid Peer-to-Peer** system and as the name suggests, the system belongs to a class of P2P systems known as *Hybrid* P2P systems which have a central server that maintains peer information and provides this information to requesting entities. Note that, the actual content is not stored on the central server but is delivered by the peers themselves. The other class, known as *Pure* P2P systems, do not have such a server and the peers themselves act as clients and servers. Although a system based on the proposed topology can be built as a pure P2P system, we believe a hybrid P2P architecture offers benefits such as scalability, security and flexibility due to centralized management. We briefly discuss our hybrid architecture.

4.1. Roles of the System

The design of our system follows the *Responsibility Driven Design* approach. The system takes on the following roles as enumerated below :

1. Supernode : A *Supernode* is the entity in the proposed hybrid P2P system that maintains peer information for every ongoing session. The delivery of some content from a source to multiple peers using the proposed topology constitutes a session. A Supernode manages many such sessions. It is a resource-rich entity which handles all requests from peers for hosting, joining or leaving sessions. It maintains an accurate global view of the topology of a session.

2. Peer : A *Peer* is an entity that is part of some session which is managed by a Supernode. This implies that it either hosts a session or joins an ongoing session managed by a Supernode. A peer is connected to other peers in its session as instructed by the Supernode to form the proposed topology. At any point in time, a peer only knows about its neighbors and its sources. Any changes in the topology are conveyed to it by the Supernode through standard messages discussed later. Once all participating peers are connected in the desired manner, they are responsible for the delivery of the content amongst themselves. The source starts disseminating data to its immediate neighbors, and eventually, with assistance from all participating peers, the content is delivered to and received by all the participating peers as mentioned earlier.

3. In addition to these main roles, a *Node* can be an entity only interested in obtaining information about all the ongoing sessions on a published list of Supernodes. It is not a part of any session and hence, not a part of the P2P network until it decides to join or host a session after which it becomes a *Peer*.

4.2. Subsystems Involved

The entire Hybrid P2P system can be classified into various subsystems as enumerated below :

1. *Peer subsystem* which is concerned with functioning of the peers in the proposed system. It deals with sending requests, handling responses and the actual content delivery. This subsystem is discussed in detail in Chapter 6.

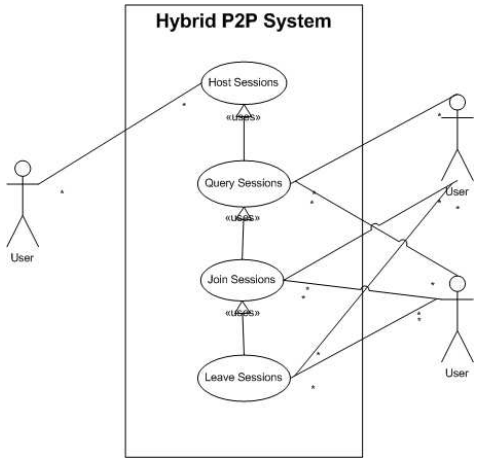


FIGURE 4.1. *Use Case Diagram*

2. *Supernode subsystem* which is concerned with Supernode functionality. It is out scope for this work. [32] deals with the Supernode subsystem in detail.

3. Common subsystems like the *Wrapper subsystem* and the *Message subsystem* which are used by both the above subsystems. The common subsystems are discussed in Chapter 5.

4.3. Scenario Walkthroughs

Next, we briefly describe a few scenarios faced by the system as shown in Figure 4.1, and the interaction between a *Peer* and a *Supernode* in each of these scenarios.

4.3.1. Querying Supernodes

A *Node* may obtain information of ongoing sessions at various Supernodes by querying them. Supernodes, in turn, process the query and respond by sending back, to the node, a list of sessions managed by it. The node, upon receiving the response, processes the information and is responsible for displaying it to the user in a meaningful manner.

4.3.2. Hosting Sessions

A *Peer* may choose to host a session. It sends an appropriate request to an arbitrarily chosen *Supernode* from the published list of Supernodes it possesses. The request message contains information such as the file to be streamed and the preferred transport protocol. The Supernode, upon receiving this request, determines if it has the resources to manage the new session. If it is capable of hosting the new session, it allocates the necessary resources to create a new session and generates a unique session ID for the new session. It sends a positive response containing the unique identifier for the new session back to the peer. Upon the receipt of the positive response, the peer allocates resources for the new session at its end and begins streaming data. The new session is now published at the Supernode and is available to other peers interested in receiving the content.

4.3.3. Joining Sessions

Recall that, a peer obtains a list of sessions managed by a Supernode by querying the Supernode. If it elects to join an existing session, it sends a join request to the Supernode. The request contains the unique session ID that identifies

the session that the new peer has chosen to join. When the Supernode receives the join request from the peer, it updates the mesh topology to accommodate the new peer. The update is a two-stage process. First, existing peers have to be updated so that they can find their new neighbors. Second, the new peer needs to know about the peers it needs to connect to, in order to be a part of the mesh. The Supernode notifies each of the affected peers about the changes in their positions in the mesh topology for the session. A peer, upon receiving an update message takes the necessary action by making new connections and/or discarding invalid connections. Once the changes are tested out, the peer sends back an acknowledgement message informing the Supernode that the updates were a success. Upon receiving such acknowledgements from all affected nodes including the new peer, the Supernode finalizes all the updates and mesh topology for the session changes permanently. It also notifies all affected peers to finalize their changes and finally, the mesh is fully updated.

4.3.4. Leaving a Session

When a peer no longer wants to be a part of an active session, it requests the concerned Supernode for permission to leave the session. The Supernode updates the session mesh by taking action as described above. Once the mesh is in a stable state after changes have been carried out, the Supernode grants permission to the departing peer via an acknowledgement message. This process ensures that a departing peer does not cause disruptions to the streaming session by creating temporary disconnections.

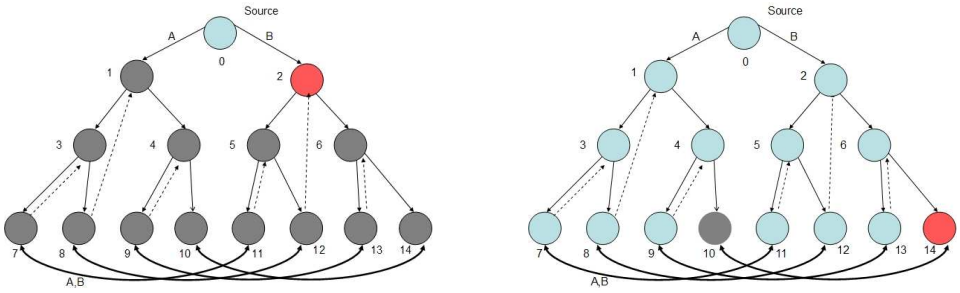
4.4. Features of the System

Some of the features of the System are mentioned below :

1. Heartbeats : We mentioned that if a peer wishes to leave a session, it must inform the Supernode. However, it is possible that a peer may leave without informing the Supernode. If such cases are not detected, it may cause other peers to face data loss. Therefore, to counter such cases, we incorporate a *Heartbeat* functionality. All peers periodically send out heartbeats to their neighbors. Each peer also maintains a list of sources that it gets data from. If a peer does not receive a heartbeat from any one of its sources for a stipulated period of time, the source peer is declared to be dead. The peer informs the Supernode about the failed peer and Supernode rearranges the mesh and brings it to a stable state as described in Section 4.3.4. Assuming the timeout interval is sufficiently low, the data loss before the detection and maintenance of the mesh is very little.

2. Optimization : In Section 1.4, one of the assumptions made was that the upload capacities of all participating peers would be similar. However, for all practical purposes, this may not necessarily hold true. Even so, the assumption that, the bottleneck is the upload capacity of a peer, still holds. Therefore, we have implemented an optimized version of the system that ensures that in any given situation, the system performs as best as it can. The peers are classified based on their connection types (T1, DSL, etc.). It is evident that if a low upload capacity peer is at a high level in the mesh structure, peers in its group and their siblings would be affected. In contrast, if such a peer is placed as far down the mesh as possible, fewer peers would be affected. The optimized system achieves this by swapping

potentially bad peers that are higher up in the mesh with better peers as and when they join the session resulting in the bad peers ending up towards the bottom of the mesh and affecting a relatively lesser number of peers. Consider the non-optimized topology in Figure 4.2 (a). Assume that peer 2 relatively has a very low upload capacity. If it is placed at a high position in the mesh, it affects all the other peers in the mesh. As opposed to this, consider the optimized mesh in Figure 4.2(b) where constant swapping with better peers results in peer 2 taking the place of peer 14 as the rightmost peer in group 2. From this position in the mesh, the low upload capacity peer 14 affects only one other peer i.e. peer 10. Thus, the overall system throughput is much higher than that in the non-optimized mesh.



(a) (b)

FIGURE 4.2. *(a) Non-optimized Mesh (b) Optimized Mesh*

4.5. Advantages of the System

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

- Load Distribution : Since the design separates topology management and data dissemination, the peer subsystem focuses on the delivery of the content as instructed by the Supernode. The complexity of running the algorithm and any issues related to optimizing the performance of a session are handled by the Supernode subsystem.

- Security : Having centralized control over the influx of peers into the system prevents malicious user activity. The Supernode can authenticate the peers participating in a session. As an example, suppose a confidential live meeting is streamed over the corporate network intended for a selected audience, the Supernode handling this session can authenticate peers as they join the session.

- Flexibility : Hybrid architecture offers high flexibility in terms of management and upgrades. Any major changes related to the data dissemination technique can be carried out at the Supernode without affecting the rest of the system.

4.6. Drawbacks of the System

Currently, the Supernode is a single point of failure. If the Supernode managing a session fails, the existing peers in the session can continue communicating and delivering content. However, peers looking to join or leave the session would be unable to do so. This drawback can be overcome by ensuring that there are multiple Supernodes across which session information is smartly replicated. If a Supernode were to fail, some other Supernode could detect its failure and it could easily take over and ensure the smooth functioning of the ongoing sessions managed by the failed Supernode.

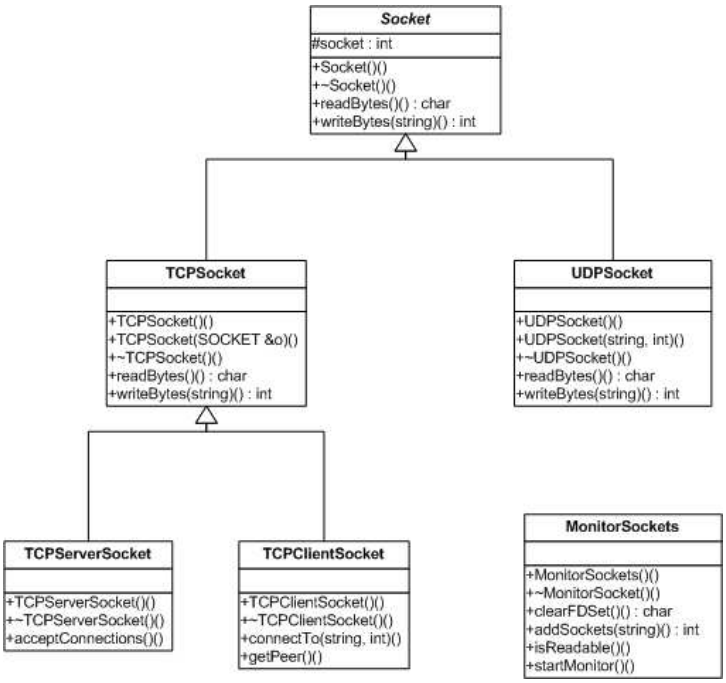


FIGURE 5.1. *Class Diagram - Wrapper Library*

5. COMMON SUBSYSTEMS

The major subsystems like the *Peer* subsystem and the *Supernode* subsystem make use of common subsystems like the *Socket Wrapper* subsystem and the *Message* subsystem to communicate with each other. In this chapter, the common subsystems have been described.

5.1. Socket Wrapper Subsystem

As shown in Figure 5.1, we have our own C++ wrapper classes to perform basic network operations such as socket creation, reading from and writing to sockets and so on. Some of the wrapper classes are :

1. Socket Class : It is the base class containing a socket descriptor. All other types of Socket classes are derived from the Socket class.

2. TCPSocket Class : This class is derived from the base Socket class and it represents TCP Sockets. It contains the methods for writing to and reading from TCP Sockets.

3. UDPSocket Class : It represents another class of sockets known as UDP sockets and is also derived from the base Socket class. This class contains the implementations for the socket read and write operations for UDP sockets.

4. TCPServerSocket Class : This class represents a special type of TCP sockets that are concerned with listening for, accepting and servicing incoming requests from clients. It is derived from the TCPSocket class. This class binds the socket created to a specified port and listens for incoming connection requests. It also contains a method, *accept()*, that is capable of accepting incoming connection requests and assigning sockets to the new connections for all further communication between the two entities.

5. TCPClientSocket Class : It is also derived from the TCPSocket class. This class represents the clients capable of connecting to listening sockets, sending requests and handling responses. It contains a method, *connectTo()*, that connects to a listening socket on the specified host at the specified port.

6. MonitorSockets Class : Finally, we also have a class whose role is to monitor a group of sockets for any kind of activity. We need such a monitoring feature as the Peers and Supernodes employ extensive socket multiplexing. This class basically encapsulates the *select()* function that is capable of monitoring such multiplexed sockets.

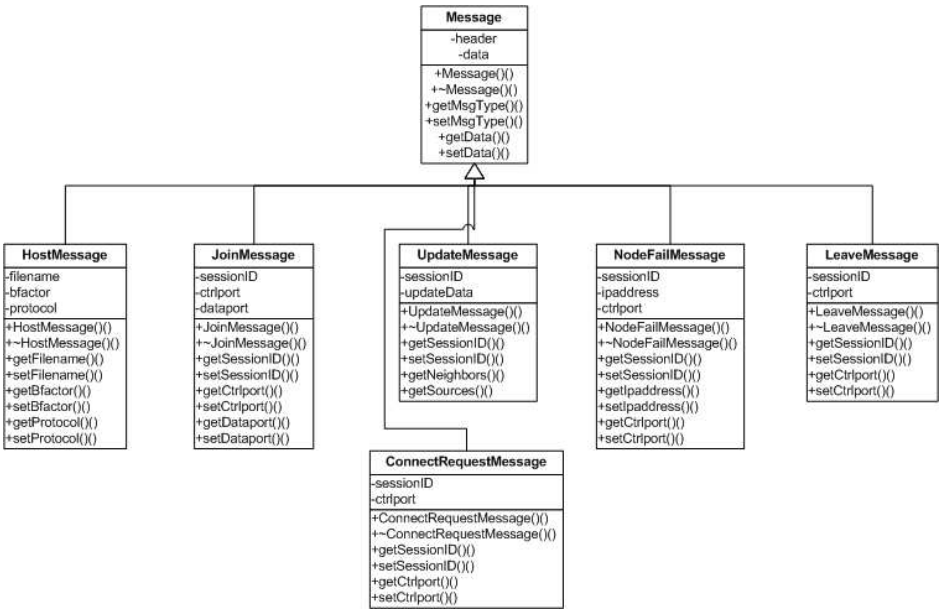


FIGURE 5.2. *Class Diagram - Message Library*

5.2. Message Subsystem

Figure 5.2 shows the hierarchy of the Message classes. They are described below :

1. Message Class : It is the base class from which all other message classes are derived.

2. HostMessage : It is derived from the Message class and is used by a peer to indicate to a Supernode that it wishes to host a session.

3. JoinMessage : It is also derived from the Message class and is sent by a peer to a Supernode when the peer wishes to join an ongoing session.

4. LeaveMessage : It is sent by a peer to a Supernode to indicate that the peer intends to leave an active session.

5. NodeFailMessage : It is sent by a peer to a Supernode whenever a failed node is detected.

6. ConnectRequestMessage : It is sent from one peer to another requesting permission to make a connection.

7. UpdateMessage : It is sent by a Supernode to a peer if the position of the peer changes due to peers joining or leaving sessions.

8. AckMessage : It is sent from a Supernode to a peer as a positive response to any request. It may contain additional data members depending upon the type of acknowledgement it carries.

9. NackMessage : It is sent from a Supernode to a peer as a negative response to a request by the peer. It contains additional information such as an predefined error code.

In addition to the data members contained in each of these classes, they also contain the following methods:

1. *makeMsg()* : This method converts the corresponding message object according to the protocol into a form that can be sent over a network.

2. *parse()* : Each message object invokes this method to parse the message sent over a network as described above into an appropriate form.

Thus, it is evident that the rest of the system need not know about the protocols used for communication between peers and Supernodes. All of these messages along with a brief description of their fields are described in Chapter 6 as we work through various scenarios.

6. PEER SUBSYSTEM

In this chapter, the Peer Subsystem is described in greater detail. The Peer Subsystem follows a *Responsibility Driven* design and comprises of various components each of which carries a certain responsibility. The Peer subsystem interacts with the Supernode subsystem via messages which will also be included in the discussion that follows.

6.1. Components of the Peer Subsystem

In this section, we describe the various classes that make up the Peer subsystem (Refer Figure 6.1). The description includes the important attributes and methods exposed by each of these classes.

6.1.1. Node Class

The Node class acts as the base class and represents users who wish to query Supernodes for ongoing sessions. Thus, an important method of the Node class is :

1. querySupernodes() : A Node has access to a published list of Supernodes that are always alive. This method queries each Supernode in the list and downloads a list of ongoing sessions managed by each of the queried Supernode. It does this by sending a ListSessionRequest message as shown in Figure 6.2. A node identifier is a combination of the IP address of the peer and its control port on which all control data is received. To this, the Supernode responds with a ListSessionsResponse message as shown in Figure 6.3 which contains a list of all the sessions managed that Supernode.

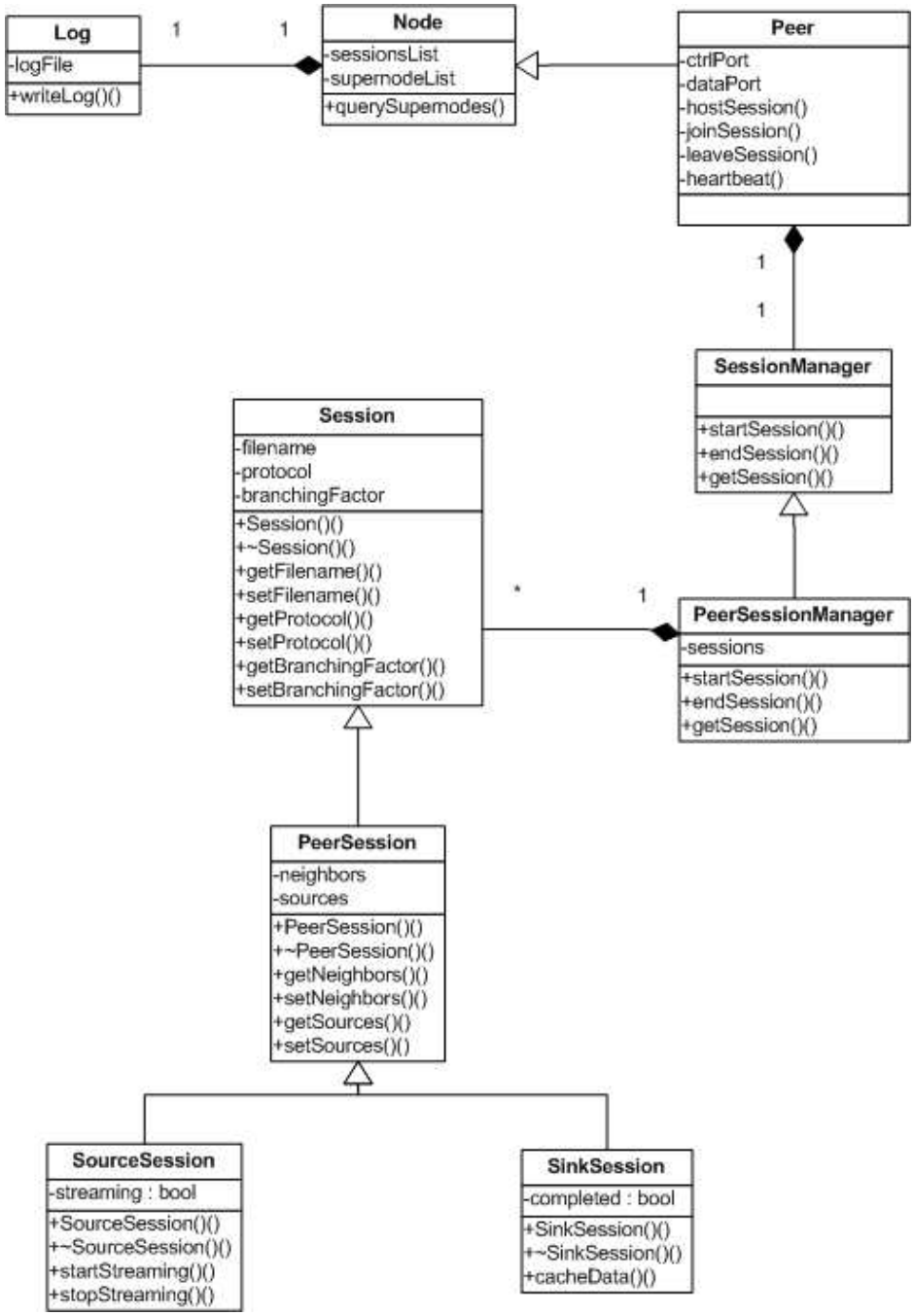


FIGURE 6.1. *Class Diagram : Peer Subsystem*

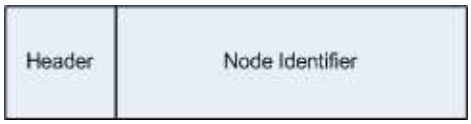


FIGURE 6.2. *Structure of a ListSessions Request*

FIGURE 6.3. *Structure of a ListSessions Response*

6.1.2. Peer Class

The *Peer* class is derived from the Node class and is the main component of the Peer subsystem. Its responsibilities include allowing users to host sessions, join ongoing sessions and leave active sessions. Some key data members are listed below :

1. ctrlListeningPort : This is the port on which a peer listens for incoming control information. Control information for all sessions arrives on this port.

2. dataListeningPort : A peer receives data packets on this port. A peer has a common port for incoming data packets for all sessions.

Some of the important methods of the Peer Class are :

1. hostSession() : The peer sends a HostSessionRequest message to an arbitrarily chosen Supernode from the list of Supernodes. the structure of the HostSessionRequest is as shown in Figure 6.4. The Supernode responds

with either a HostSessionResponse containing a generated unique Session ID (Figure 6.5), or a negative acknowledgement (NACK) (Figure 6.6). The unique key field is used by the Peer to identify the response for a HostSessionRequest. Upon receiving the session ID, the Peer Class invokes one its components, the PeerSessionManager, to allocate the necessary resources. The PeerSessionManager allocates a SourceSession object from the pool of available session objects and the peer is now ready to stream content to other peers who join the session.

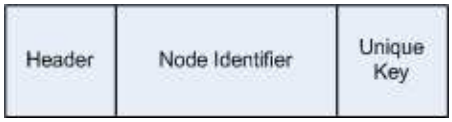


FIGURE 6.4. *Structure of a HostSession Request*

FIGURE 6.5. *Structure of a HostSession Response*

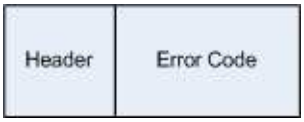


FIGURE 6.6. *Structure of a NACK*

2. joinSession() : The peer sends out a JoinSessionRequest to the Supernode in charge of the session that the peer wishes to be a part of. The structure of a JoinSessionRequest is as shown in Figure 6.7. The Supernode responds with

either a NACK or an UpdateMessage (Figure 6.8)that contains instructions to assist the peer in joining the mesh at the appropriate place. The data section of the UpdateMessage contains the following :

(a) Connect Messages : These messages (Figure 6.9) enable the peer to join the mesh. MsgType maybe a *C* indicating that the peer should connect to the IP address specified in the next section of the Connect Message. The tag ID field is used for forwarding data packets as described later on.

(b) Disconnect Messages : These are used with MsgType set to *D* to instruct the peer to sever any connections with a peer at the specified IP address. Refer Figure 6.10.

(c) InputLink Messages : These mesages (Figure 6.11) provides information needed for the Heartbeat feature described later in the section. The Action field is either a *A* indicating that the peer details are to be added to the list of sources or an *R* causing the specified peer to be removed from the list of sources.

Once again, a SinkSession object is allocated for the new session in conjunction with the PeerSessionManager component.

FIGURE 6.7. *Structure of a JoinSession/LeaveSession Request*

3. leaveSession() : This method is invoked when a peer intends to leave a session that it is a part of. The mechanism is similar to that of a joinSes-

FIGURE 6.8. *Structure of a UpdateMessage*

FIGURE 6.9. *Structure of a Connection Message*

FIGURE 6.10. *Structure of a Disconnection Message*

FIGURE 6.11. *Structure of a InputLink Message*

sion() invocation. A peer sends out a leave session request to the concerned Supernode. The fields of this message are identical to that of the joinsession request message (refer to Figure 6.7). The Supernode accordingly updates the mesh and notifies all affected nodes. However, after all update messages have been sent out by the Supernode and acknowledged by the affected peers, the Supernode sends out a leave request acknowledgement to the departing peer. Only after receiving this acknowledgement from the Supernode, may the peer deallocate the session resources by invoking its PeerSessionManager object and leave the session.

4. Peer() : The constructor method creates a TCPServerSocket object, *ctrlListener* and a UDPSocket object *dataListener*. It binds both these objects to the *ctrlListeningPort* and the *dataListeningPort* respectively. It also invokes the *startMonitoring()* method to start a thread that monitors for incoming data or control information.

5. startMonitoring() : This protected method is invoked through the constructor of the Peer class. This method spawns a thread which is responsible for monitoring all the open connections that a peer may have.

6. doMonitoring() : This method implements the thread for monitoring the *ctrlListeningPort* and all open connections for incoming control information and data packets. It creates the MonitorSockets object from the Wrapper library for monitoring all open connections. Initially, a peer has two objects to monitor viz. the *ctrlListener* and the *dataListener*. When activity is detected for the *ctrlListener*, the peer recognizes it as a request for a connection. It uses socket multiplexing and allocates a TCPClientSocket object for the new connection request. This new socket is in charge of the servicing

the new request and it is also added to the list of objects that need to be monitored by the peer. If activity is detected on any of these objects, the message is extracted and a message handling routine is invoked which then takes over as described next.

7. handleMessage() : This method is invoked whenever control information or data is received by a peer. The message handling routine is responsible for parsing the information or data received and invoking the appropriate methods of the sessions for which the information or data was meant for. It does the parsing by creating a *Message* object with the raw information. The header of the message object determines what type of a message was received. An object of the specific message type is created which then parses the various fields of the message.

The Peer Class is assisted in its responsibilities by various other components whose descriptions follow.

6.1.3. SessionManager Class

The *SessionManager* class is the base class of the *PeerSessionManager* component which assists the Peer class in the management of the pool of session objects. An important data member of the SessionManager class is *maxSessions* which specifies the maximum number of sessions that a peer can be a part of. It also contains the virtual methods *createSession()* and *closeSession()* that any derived classes such as the PeerSessionManager need to override and provide implementations for.

6.1.4. PeerSessionManager Class

The *PeerSessionManager* class is derived from the SessionManager class. Its main task is to assist the Peer class in allocating resources for the sessions that the peer is a part of. It uses the derived data member, *maxSessions*, to maintain a pool of session objects that are allocated or deallocated when the peer hosts/joins or leaves sessions respectively. The pool of session objects is also organized into lists of free, reserved and active sessions. Some of the available methods of the PeerSessionManager class are :

1. createSession() : This method is invoked when a peer either hosts or joins a session. The pool of session objects is examined for an available resource and if found, it is assigned to the new session. The session object maybe a *SourceSession* object if the peer is hosting a session or a *SinkSession object* otherwise.

2. closeSession() : This method is invoked when a peer intends to leave a session. Specifically, it is invoked after the peer has received an acknowledgement from a Supernode for a leave request sent by the peer as described earlier. The specific session object is freed up and is now available for allocation if needed.

3. reserveSession() : This method is invoked before a peer sends out a join session or host session requests. This reserves a session object from the pool for the new session before the request can be sent out. It does this by fetching a free session object and then adding it to the list of reserved sessions that it maintains internally. This scheme ensures that a peer does

not run out of resources when it receives a response to any of its requests for a new session viz. the join session request or the host session request.

4. getFreeSession() : It returns a free session object for reservation from the list of free session objects that it maintains. If there is no such object in the free sessions list, it looks through the list of reserved sessions to check whether any of reserved sessions have not been claimed for long periods of time. If such a session object exists, it is reclaimed as a free session and returned.

6.1.5. Session Class

The *Session* class is the base class from which classes like the *PeerSession* class are derived. The important data members of this class are :

1. sessionID : The identifier used by the Supernode and the peer to refer to a particular session.
2. protocol : The protocol used by the session (TCP or UDP).
3. fileName : The name of the file being streamed.
4. branchingfactor : The branching factor b of the session.

In addition, the Session class also contains accessor methods for the above mentioned data members.

6.1.6. PeerSession Class

The *PeerSession* class is derived from the *Session* class and therefore, inherits all the data members and methods of the latter. In addition, the PeerSession class contains the following data members :

1. supernodeIP : The IP address of the Supernode managing the session.
2. supernodePort : The port number of the listening socket on the Supernode.
3. Neighbors : The list of peers that it needs to forward incoming data to. It is represented by an STL, *multimap*, where the key is the tag ID.
4. Sources : The list of peers that it expects to receive data from. This data member is used by the *Hearbeat* feature to detect failed nodes. The data type used is the STL *map*, where the key is the source information. Each source is also associated with a timestamp which records the time of the last received heartbeat from that source.

The important methods of the PeerSession class are :

1. updateConnections() : This method is invoked whenever a peer receives an update message from a Supernode for this session. In this method, the updated information of neighbors and sources sent by the Supernode is stored at temporary locations. Recall that all updates are made permanent only after instructed by the Supernode via a finalize message.
2. doUpdate() : This method is invoked after a peer receives a finalize message from the Supernode. This method results in all the above updates being made permanent.

3. heartbeatproc() : This method basically implements the *Heartbeat* feature of the system. It implements a thread that periodically sends out heartbeats to its neighbors. The thread also monitors the list of sources for their heartbeats. If a failed peer is detected, a NodeFail message (Figure 6.12) is sent to the Supernode.

FIGURE 6.12. *Structure of a NodeFail Message*

4. sendConnectRequest() : This method is invoked when an update message containing new neighbors is received. All of the new neighbors are requested for permission to establish connections. This prevents the situation where the new neighbors are dead or unable to accept any more connections.

5. handleConnectResponse() : This method keeps a track of acknowledgements received from peers in response to the requests for permission to connect sent in the above method. As soon as a positive response is received from all the expected peers, this method implicitly sends an acknowledgement message to the Supernode informing it that the changes in the Update message were a success.

6.1.7. SourceSession Class

This class is derived from the *PeerSession* class. This class represents peers that host a session. In addition to the inherited members, it contains the following methods :

1. startStreaming() : When this method is invoked, the peer starts streaming data to its neighbors. For streaming, it employs the data partitioning scheme as described in Chapter 3. It implements a spawned thread to stream data. The source file is read as fixed size blocks using which data packets are created. Among other fields, tag IDs are generated for the data packets and each data packet is delivered to the appropriate neighbor based on the tag ID matching. Figure 6.13 shows the structure of a data packet.

FIGURE 6.13. *Structure of a Data Packet*

2. stopStreaming() : The invocation of this method causes the streaming of data to cease. Note that, the session is still alive and the source may start streaming by invoking the previous method.

6.1.8. SinkSession Class

This class, like the *SourceSession* class, is derived from the *PeerSession* class and represents peers that join an ongoing session in order to receive the content being distributed. It inherits all the members of its parent class and includes an additional data member, *buffer*. This is used for buffering about 6 secsonds of encoded video data. For future work, this buffer will be read, decoded and fed to a media player. The SinkSession class also contains the following methods :

1. cacheData() : This method is internally used by the transmitData() method described next and is used to cache incoming data. While storing the incoming data packets, reordering is also done as the data packet contains sufficient information to allow for this. Refer to Figure 6.13.

2. transmitData() : This method is invoked whenever a session data packet is received by a peer. The peer caches the data in the buffer and then forwards it to its neighbors in the session mesh. Recall that the neighbors are stored in the multimap along with a tag ID field. A data packet is forwarded to neighbors with identical tag IDs as the data packet itself. The working of the content delivery mechanism is described in Section 6.4.

6.1.9. Log Class

This class essentially represents the logging component of the Peer subsystem. It contains a data member, *logFile*, that represents the physical file on the disk where key events would be logged. Additionally, the Log class contains a method, *writeLog()*, which takes a parameter of type, *LogRecord*, and writes the record to the log file. The LogRecord structure is as shown in Figure 6.14. The field *logType* is a predefined code for an event. Examples are HostRequest event or the JoinRequest event. The *timestamp* field records the precise time of the event. *Identifier* field contains a value that identifies the event. For example, for a JoinRequest event, the session ID is used as the identifier. *Params* field is optionally used to record additional information. For instance, while logging events to measure the node delay, it is useful to store the packet sequence number as additional information. The user defined structure also contains a method,

toString(), that converts all the above fields into a string that can be written to the log file.

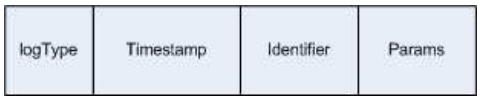


FIGURE 6.14. *Structure of a LogRecord object*

6.2. Interaction Diagrams for Key Events

In this section, we present the interaction diagrams for the scenarios mentioned in Section 4.3.

6.2.1. Querying Supernodes

As depicted in Figure 6.15, a *Node* interested in obtaining the lists of ongoing sessions across various Supernodes can do so by a sending out a *ListSessionRequest* message to each Supernode. It has access to a publicly available list of Supernodes and can refer to this list to access all the Supernodes. Upon receiving this request, a Supernode responds with a message, *ListSessionsResponse*, that contains information about the sessions managed by it. It is the responsibility of the *Node* to cache it and display it to the user in a meaningful manner. Logging for statistics is also carried out in the process.

6.2.2. Hosting a Session

As shown in Figure 6.16, to host a session, a peer sends a *HostSessionRequest* message to a Supernode. The Supernode replies with an acknowledgement

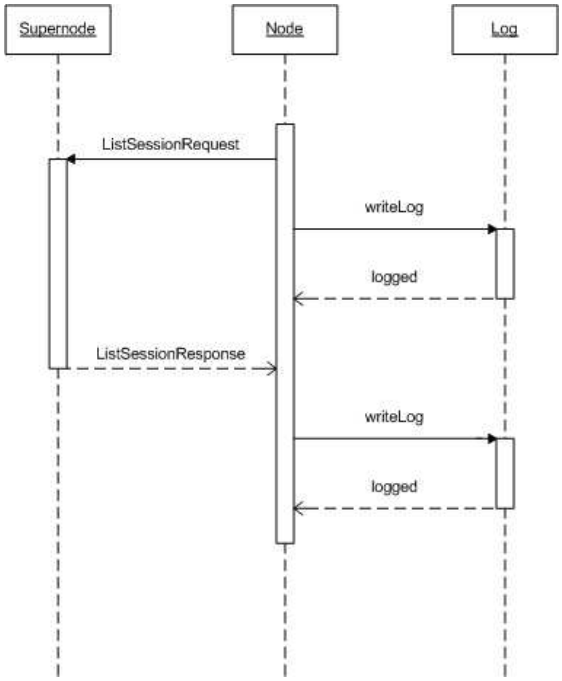


FIGURE 6.15. *Sequence Diagram for Querying Supernodes for Ongoing Sessions*

containing a generated unique session ID. The peer allocates resources, in the form of a *SourceSession* object, in conjunction with the *PeerSessionManager*. All the events are logged for statistics as shown. Now, the peer acts as the source and can start streaming data as other peers join in.

6.2.3. Joining a Session

Figure 6.17 depicts the sequence of events that are involved when a peer wishes to join an ongoing session. Assume that the peer has the session ID of the session it wishes to join. It sends a *JoinSessionRequest* to the associated Supernode containing the session ID. The Supernode responds with an *Update* message, as described in Section 5.2, containing instructions that enable the peer

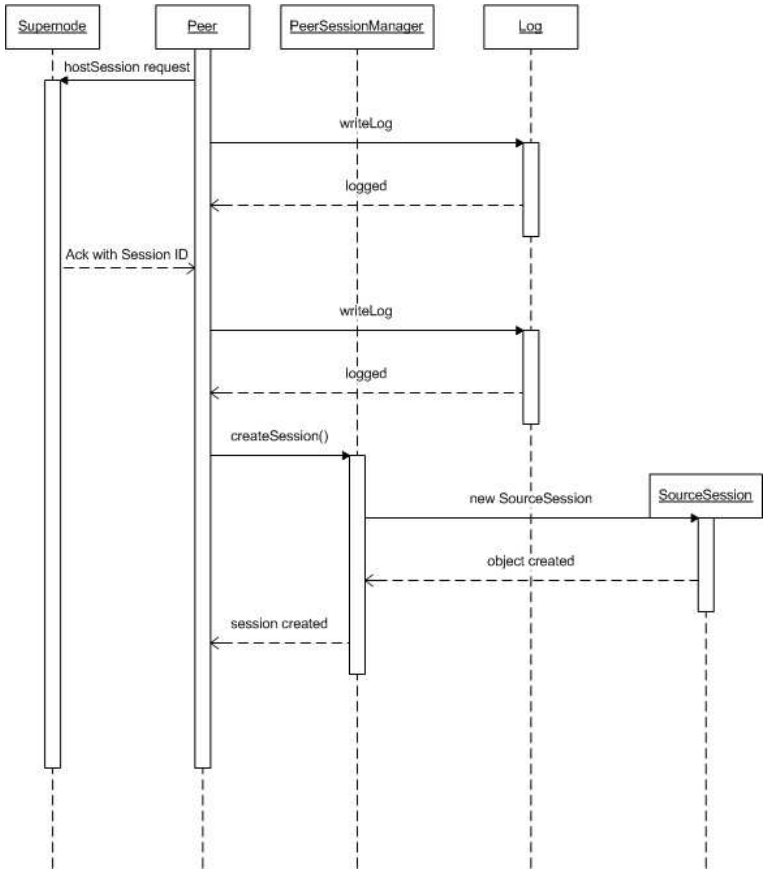


FIGURE 6.16. *Sequence Diagram for Hosting a Session*

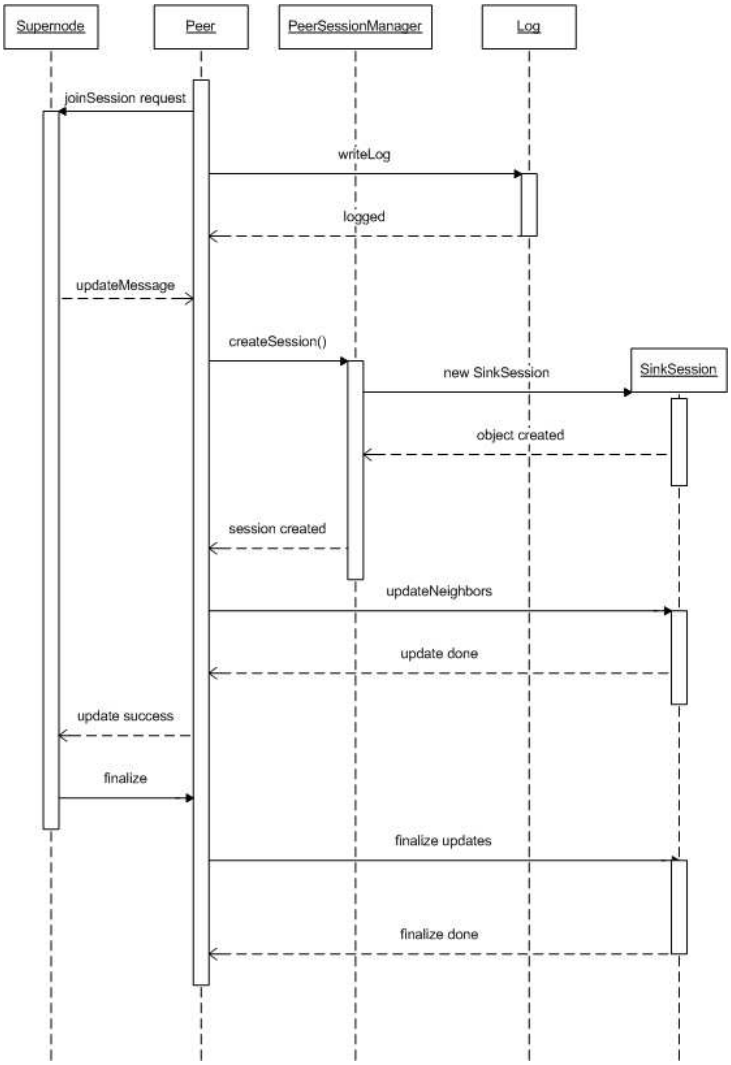


FIGURE 6.17. *Sequence Diagram for Joining an Ongoing Session*

to become a part of the session. Resources are allocated as described above and using the information contained in the update message, connections to other peers are made. After all connections have been successfully made, the peer sends out an acknowledgement to the Supernode. The Supernode responds with a *Finalize* message which instructs the peer to make all changes permanent. Now, the peer is a part of the session and will start receiving the content being distributed. All key events are logged as described above.

6.2.4. Leaving a Session

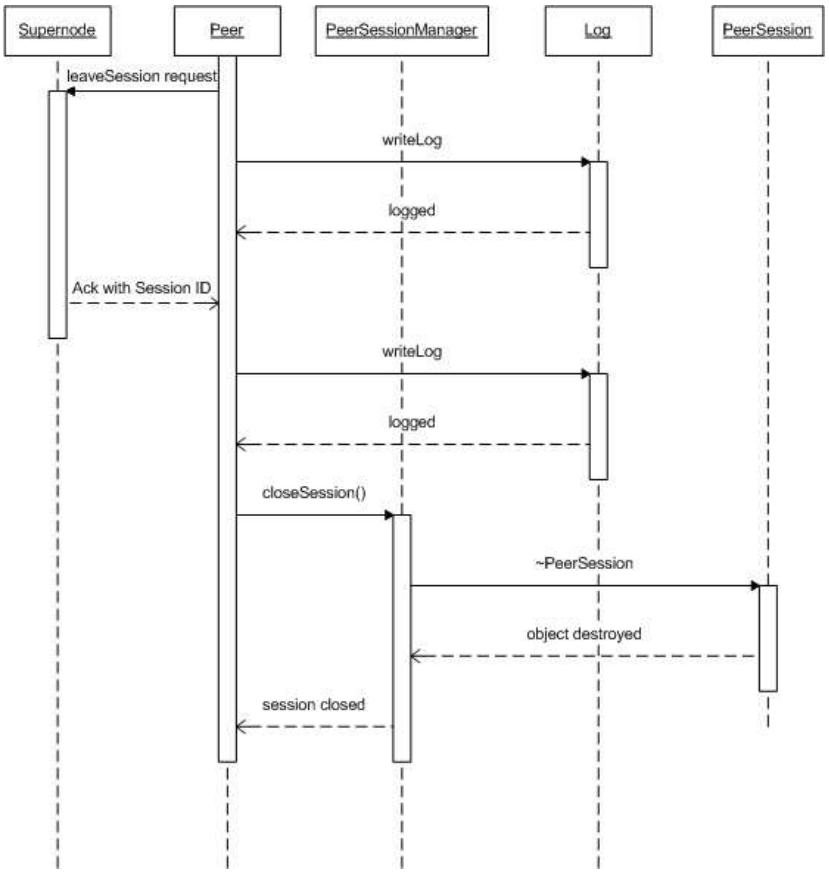


FIGURE 6.18. *Sequence Diagram for Leaving an Active Session*

We see, in Figure 6.18, that when a peer intends to leave an active session, it must send a *LeaveSession* request to the associated Supernode. The Supernode undertakes the reconstruction of the mesh and signals the success by sending an acknowledgement to the peer. Upon receipt of this acknowledgement, the peer may leave the session. Note that, the peer is very much a part of the session until the acknowledgement is not received. Thereafter, the resources are freed and the peer is no longer a part of the session.

6.3. Flow of the System

In this section, we describe the general flow of the Peer subsystem and mention some of the key activities involved. Note that, this infrastructure would be used as a library on top of which a GUI-based application could be developed.

1. The role of the system, by default, at application startup is that of a *Node*. An object of the *Node* class requires much lesser resources than that of the *Peer* class. Thus, if a user is not certain he wants to join or host a session, a Node object would suffice.

2. To query Supernodes, the Node object seeks a file, *server.dat*, specified as a parameter to its constructor. This file is assumed to be widely available similar to the *server.met* file for the *EDonkey* [9] and *EMule* [10] applications. *Server.dat* contains a list of Supernodes that are always alive and managing sessions. The application establishes TCP connections to each of the Supernodes listed in the file and sends a request to obtain a list of sessions managed by them. In response, every Supernode sends back a message containing a list of sessions managed by them. This raw message is in

an encoded form and stored locally in a file, *sessioncache*. It is then parsed by the Node object appropriately and displayed to the user in a meaningful manner.

3. For all other purposes, the role of the application changes to that of a *Peer* and therefore, an object of *Peer* class is created. The constructor of the Peer class sets up a *ctrlListener* and a *dataListener*. Note that, the system currently uses TCP for all control information and UDP for the actual content delivery. *CtrlListener* essentially is a TCP server socket bound to a port, specified as an argument to the constructor method of the Peer class, that listens for incoming connection requests. As soon as a new connection request by a peer or node is detected, the *ctrlListener* assigns a TCP socket to the new request accepting the connection. All further communication between the requesting entity and the peer now takes place with the newly created socket. Thus, the peer now also needs to monitor the newly created socket. Similarly, *dataListener* listens for incoming data packets. All messages and data packets are extracted and sent to the message handling routine. All of these monitoring functions are carried out in a monitoring thread created by the constructor so that the peer may focus on other activities. In addition to this monitoring activity, the peer also constructs a *PeerSessionManager* object which, in turn, manages the pool of session objects. If a peer needs to allocate resources, it must request the PeerSessionManager object to perform the task.

4. Now, suppose the peer wants to join a session. Before a request can be sent out, a peer must make sure it has the necessary resources. It makes a request to the peer session manager to reserve a session object for the new

session. The peer session manager invokes the *reserveSession* method which reserves a free session object. Internally, an object is picked from the list of free sessions. If this list is empty, then attempts are made to reclaim a reserved session object that has not been used for long periods of time. Assuming the process of reserving a session object was successful, a join request is constructed with the necessary information and delivered to the concerned Supernode.

5. Consider the case where the peer wants to host a session. Similar in nature to the previous case, the peer requests the peer session manager to reserve a session object for the session to be hosted. If a session object is successfully reserved, a host session request is constructed with the necessary information and sent across to an arbitrarily chosen Supernode. In both case, after the request is sent to the Supernode and its response is pending, the peer may carry out other activities in the meanwhile.

6. Additionally, a peer might want to leave an active session. In this case, a leave request is constructed with the session ID and the identifying information of the departing peer. This leave request is then sent to the Supernode managing the active session the peer intends to leave.

7. Another special case is that of a peer detecting a dead source. It constructs a node fail message by including the information of the dead source and sends it to the Supernode which takes appropriate action to bring the session mesh to a stable state.

8. After any of the above requests are sent, a Supernode may respond after an indefinite amount of time. However, whenever a peer receives control

information, it relays the incoming information to a message handling routine. The task of this routine is to parse the incoming message, determine its type and then take appropriate action. Some of the messages and the action taken are described as follows.

(a) In case of a host session response, the message is parsed and and the peer session manager creates an object of the *SourceSession* class with the session ID obtained by the peer from the response message. Once the *SourceSession* object is created, the peer may start delivering content. A detailed explanation of the content delivery mechanism is provided in the next section.

(b) In the case of an update message, the message is parsed by the message handling routine. A reference is obtained to the appropriate session object via the peer session manager component. Finally, the concerned session object is requested to update its neighbors and sources according to the instructions contained in the update message. During the updating of the neighbors, the newly added neighbors are requested for permission to connect. Once all the new neighbors grant permission, the peer sends back an acknowledgement to the Supernode letting it know that the updates were successful. Note that, the updates thus far are stored in temporary locations. Upon receiving acknowledgements from all the peers that received the update message, the Supernode sends out a finalize message to each of them.

(c) If a finalize message is received, all the above changes mentioned are made permanent and the mesh structure changes permanently.

(d) If the parsed message is a leave response message, the peer simply requests the peer session manager to close the session which frees up all the used resources for that session.

(e) If the parsed message is a heartbeat from one of the sources of the peer, the appropriate session object is invoked which handles the heartbeat message. Typically, the new timestamp of the heartbeat message would be recorded. A connect request message is also handled in a similar manner. Recall that, a connect message is sent from one peer to another if the former peer has been instructed to deliver data to the latter.

9. For the heartbeat functionality, as soon as a peer becomes a part of an ongoing session, it starts a thread for sending and monitoring heartbeats. The thread executes at regular intervals and does the following activities in the following order :

   (a) The thread fetches the list of neighbors of the peer. A neighbor of a peer is an entity to whom the peer is required to forward data to. The thread sends out a heartbeat message to each of the neighbors at these intervals.

   (b) Next, the peer fetches the list of sources of the peer. A source of a peer is an entity from which the peer must receive data. For every such source, the thread then determines whether the timestamp of the last received heartbeat exceeds the timeout value. The discovery of any such source implies that a dead source is detected and it must be reported to the Supernode. The thread proceeds to send a node fail message to the Supernode.

10. Finally, the most significant activity for the peer subsystem i.e. content delivery is described next.

6.4. Detailed Description of the Content Delivery Mechanism

Content distribution in the proposed system is achieved using UDP sockets. Recall that, the peer object creates a *dataListener*, that accepts incoming data packets. Also, recall from Figure 6.13, along with the content, the data packet also contains information such as the session ID, hash value or the sequence number of the packet, its tag ID etc. Tag IDs for data packets are generated by the source node. On receiving a data packet, all of this information is parsed. The appropriate session object's *transmitData()* method is invoked in response to the event of receiving a data packet. This method caches the incoming data at the appropriate place so that out of order packets are rearranged. Currently, the content is cached at two places. One of them is on the disk in the form of a file. Packet reordering is carried out by writing the content at the appropriate location which can be determined by the sequence number of the packet and its data size. The other location for caching the content is an in-memory buffer that is large enough to hold about 6 seconds of video data. This is for some scheduled future work. Again, the appropriate location of the content is computed using its sequence number and data size. When the end of the buffer is reached, a wraparound takes place as we assume the initial section of the buffer has already been read by the application. After the content has been cached, the peer needs to forward the content to its neighbors. It performs the following operations to deliver the content to its neighbors :

1. It fetches all the neighbors whose tag IDs match the tag ID of the data packet. It then forwards the content to all the selected neighbors.

2. Next, it fetches all the neighbors with tag ID 0. It forwards all content to such neighbors with tag ID as 0.

This can be better illustrated with the help of an example as shown in Figure 6.19.

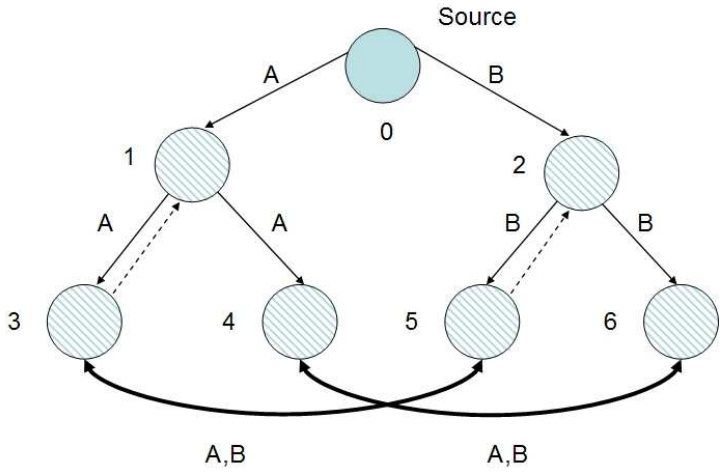


FIGURE 6.19. *Content Delivery Mechanism*

We illustrate the delivery of packet A from the source peer 0 to the rest of the peers in the mesh. As mentioned previously, tag IDs for data packets are generated at the source. Thus, for packet A, the tag ID generated is 1 and that for packet B is 2. The contents of the multimap of neighbors for the source peer 0 are as shown in Table 6.1. Since the tag ID of packet A i.e. 1 matches the tag ID of peer 1, the source transmits packet A to peer 1. At peer 1, the contents of the multimap of neighbors is as shown in Table 6.2. Now, the tag IDs of peers 3 and 4 in the multimap of neighbors match the tag ID of packet A. Therefore, packet A is transmitted by peer 1 to both peer 3 and peer 4. At peer 3, the multimap of

Tag ID	Address
1	Address of peer 1
2	Address of peer 2

TABLE 6.1. *Multimap of Neighbors for Source Peer 0*

Tag ID	Address
1	Address of peer 3
1	Address of peer 4

TABLE 6.2. *Multimap of Neighbors for Peer 1*

neighbors is as shown in Table 6.3. Based on these contents, peer 3 sends packet A across to peer 5 and similarly, peer 4 delivers packet A to peer 6. At peer 5, the contents of the multimap of neighbors is as shown in Table 6.4. Now, peer 5 delivers packet A to peer 2. Similarly, packet B is disseminated from the source to all the peers in the mesh via peer 2. In this manner, disjoint data sets are disseminated from the source to all the peers in the mesh.

Consider the topology depicted in Figure 6.20. Assume peer 7 receives data partitions A and B as described above. The multimap of neighbors for peer

Tag ID	Address
1	Address of peer 5
2	Address of peer 1

TABLE 6.3. *Multimap of Neighbors for Peer 3*

Tag ID	Address
1	Address of peer 2
2	Address of peer 3

TABLE 6.4. *Multimap of Neighbors for Peer 5*

7 is as shown in Table 6.5. Since peer 8 is listed as a neighbor with the tag ID 0, peer 7 delivers all the data partitions it receives and thus, peer 8 receives data partitions A and B.

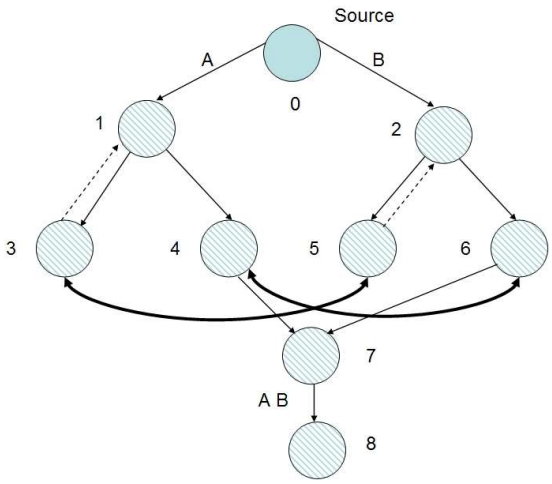


FIGURE 6.20. *Content Delivery Mechanism - contd.*

We conclude the description of the Peer subsystem with a brief description of the object oriented features.

Tag ID	Address
0	Address of peer 8

TABLE 6.5. *Multimap of Neighbors for Peer 7*

6.5. Object Oriented Concepts Explained

The design of the Peer subsystem follows an object oriented and responsibility driven approach. The reasons for employing this approach are as listed below :

1. The development of the system corresponds to what we term as programming in the large as a team is involved in this activity.

2. We understood the behavior of the system very well and therefore could work through various scenarios. Thus, the problem seemingly lent itself to Responsibility Driven Design.

3. Various components were identified and given a physical representation. We were able to view the system as a community of agents, each with a specific responsibility, interacting with one another and sharing a common objective.

4. The nature of the system suggested the use of object oriented features such as information hiding, inheritance and polymorphism to name a few.

5. Also, this is an experimental system which would constantly undergo changes such as addition/removal of features, optimization, etc. It was necessary to design components so that the coupling between them was as low

as possible. This implies that changes to one part of a system has minimal or no effect on the rest of the system.

For the reasons cited above, various object oriented concepts have been applied during the development of the system. A few of them are listed below :

- Object: An Object/Instance is an individual representative of a class. *ctrlListener, ctrlSock,* etc. and plenty of others are examples of objects.

- 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 in the system would be *PeerSession, TCPServerSocket, UDPSocket*, etc.

- Behavior and State: The behavior of a component is the set of actions that a component can perform. A *PeerSession* can add new nodes, remove existing nodes, disseminate data, etc. Its state would be described by its ID, branching factor, protocol, etc.

- Encapsulation: Storing data and functions in a single unit (class) is encapsulation. Data is not accessible to the outside world and only those functions which are members the class and the objects of that class can access it.

- Constructors: It is a procedure that is invoked implicitly during the creation of a new object value and that guarantees that the newly created object is properly initialized. When inheritance concept is utilized in the implementation the initialization of the objects of both parent and child class has to be carried out. Examples are *PeerSession()* constructor in *PeerSession* class.

- 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.

- Forward Definition: When two or more classes need to have references to each other, also known as mutual recursion, forward declaration of one of the classes is required. It just places the name in circulation leaving the completion of definition till later.

- Message Passing: It is the dynamic process of asking an object to perform a specific action.

- Inheritance: It can be defined as the process whereby one class/object acquires the characteristics of 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 may add additional functionalities. For example, the *TCPServerSocket* and *TCPClientSocket* classes inherit from the *TCPSocket* class and therefore derive the basic methods for sending and receiving data.

- Forms of Inheritance:

    1. Subclassing for Specialization: The inheritance relationship amongst the classes in the Message subsystem hierarchy exhibits this form of inheritance.

    2. Subclassing for Specification: The inheritance between *SessionManager* and *PeerSessionManager* is an example of this as the latter is required

to provide implementation of *createSession* and *closeSession* methods defined in *SessionManager*.

3. Subclassing for Extension : The relationship between the *Session* and the *PeerSession* classes is an example of subclassing for extension as the latter not only inherits the behavior of the former but adds it own functionality that may not apply to other subclasses of the former.

The benefits of inheritance such as reusability, code sharing, consistency of interface, components and information hiding can be easily observed.

- Principle of Substitution: It says that if we have two classes A and B such that class B is a subclass of class A, it should be possible to substitute instances of class B for instances of class A in any situation with no observable effect. The relationship between *SourceSession* and *PeerSession* classes exhibits this principle.

- 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.

  1. Overloading (ad hoc polymorphism): It is used to describe the situation where a single function name has several alternative implementations. The constructors *TCPSocket*() and *TCPSocket*(SOCKET &o) in the class *TCPSocket* have the same method name but different signatures and different implementations. They are examples of ad hoc polymorphism.

2. Overriding (inclusion polymorphism): It is a form of polymorphism that occurs within the context of the parent class/child class relationship. *sendBytes(), receiveBytes()* in classes *Socket* and *TCPSocket* are examples of overriding.

3. Virtual Methods: There are situations where a variable is declared as one class but holds a value from a child class and a method matching a message is found in both classes if overriding has been done. In such cases, generally, it is desirable to execute the method found in the child class. This is achieved by declaring the method in the parent as virtual. Also known as dynamic method binding, this has been extensively used in the system.

4. Polymorphic variable (assignment variable): It is a variable that is declared as one type but in fact holds a value of a different type. *PeerSession *psessions* is an example of this as it can hold either a *SourceSession* object or a *SinkSession* object depending upon the role of the peer in the session.

- Composition: A *Peer* has an instance of the *PeerSessionManager* as a data field. the responsibility of managing the pool of *PeerSession* objects has been delegated to the latter. Thus, these two display the concept of composition.

- Containers: *Vector* is an array of objects that can be as large as possible. Vectors are used extensively in the system for example to store the lists of neighbors and sources.

- STL entities such as *Multimaps*, *Iterators* and *strings* have been employed extensively in the system.

- Object Interconnections: Coupling and Cohesion provides a framework for evaluating effective use of objects and classes.

  Coupling describes the relationships between the classes. The varieties of Coupling are:

  1. Component coupling: It occurs when one class maintains a data field or value that is an instance of another class. *ctrlListener* is an instance of class *TCPServerSocket* and is a data field of class *Peer*.
  2. Parameter Coupling: Example of *Log* object as a parameter in the constructor of *PeerSession* class.
  3. Subclass Coupling: This type of coupling is visible by the usage of the concept of inheritance. An example here would class *Peer* inheriting from class *Node*.

  Cohesion describes the relationships within the classes. The varieties of cohesion are:

  1. Functional cohesion: *Socket, TCPClientSocket, TCPServerSocket, UDPSocket*, etc. are examples of functional cohesion.
  2. Data cohesion: Class *PeerSession* is an example of data cohesion.

- 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.

  1. Singleton: Classes *Node* and *Peer* are Singletons in that only one object of these classes can be created.

2. Proxy: The Message class hierarchy hides the protocol details from the rest of the system.

7. PERFORMANCE EVALUATION

7.1. Small Scale Deployment over PlanetLab

We have built our hybrid P2P system and deployed it on PlanetLab [29] nodes. An extensive set of experiments were run, the results of which are depicted and discussed below.

7.1.1. System Throughput Evaluation

In the first set of experiments, we compared the performance of vanilla multicast and our proposed system. In order to get as fair a result as possible, we ensured that the runs were conducted on the same set of machines with the peers occupying the same logical positions in our system mesh as well as in the multicast tree. To simulate the DSL upload bandwidth bottleneck, the sending rate of the source was limited to 30KBps. We calculated the average upspeed and downspeed after recording those values for all the peers.

Figure 7.1 shows the performance comparison between the vanilla multicast and our proposed mesh with 15 peers including the source. For 3 different sets of runs, the number of out-going links were $b = 2$, $b = 3$ and $b = 4$ respectively. As seen in Figure 7.1(a), the hybrid mesh outperforms the vanilla multicast in each of the runs in terms of the average downspeed. Further observation reveals that as b increases, the hybrid mesh's performance gain increases compared to the vanilla multicast. This is an expected result as, in theory, the proposed system outperforms vanilla multicast by a factor of b. The reason behind the performance gain can be attributed to the upload contribution of the leaf peers in the proposed system. This is depicted in 7.1(b). Given, the source sends out data at the rate of

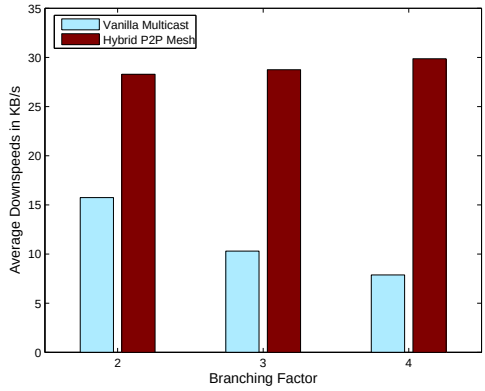
C KBps, in the case of vanilla multicast, the non-leaf peers send out useful data at a rate of C/b KBps resulting in reduced uploads as b increases. Whereas, in the case of the proposed mesh, almost all peers send out useful data at the rate of C KBps resulting in every peer achieving downspeeds of close to C KBps as opposed to C/b KBps for vanilla multicast. Therefore, irrespective of the branching factor b, the proposed mesh achieves an efficiency of close to 1.

Figure 7.1(c) compares the average file transfer time for vanilla multicast and the proposed mesh. A file of size approx. 315 KB was distributed with the source sending content at a rate of approx. 30 KBps. As seen, the transfer time for vanilla multicast increases with increase in b whereas it remains nearly constant in the case of the proposed system.

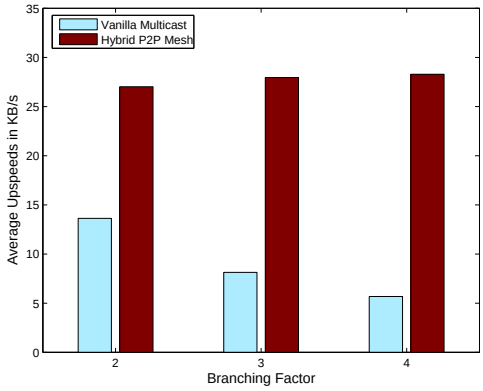
7.1.2. Packet Delay Evaluation

Recall from Section 3.4, the source transmits disparate data partitions to different groups and peers in each group end up receiving data partitions from the other groups. Experiments were conducted to measure the average time taken by a peer to receive all data partitions. The setup for this experiment comprised of 21-peer topologies with different values of b.

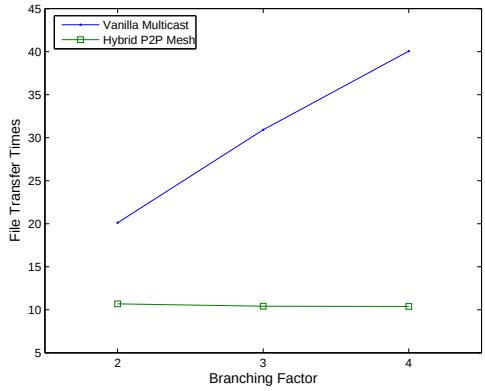
As observed in Figure 7.2, packet delay greatly depends upon the branching factor b. Therefore, we see a decrease in the delay times as b increases. This can be attributed to the fact that as b increases, the average depth of a mesh decreases. The exceptions might the peers in the secondary mesh which are, however, outnumbered by the peers in the primary mesh with lesser depth.



(a)



(b)



(c)

FIGURE 7.1. *(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*

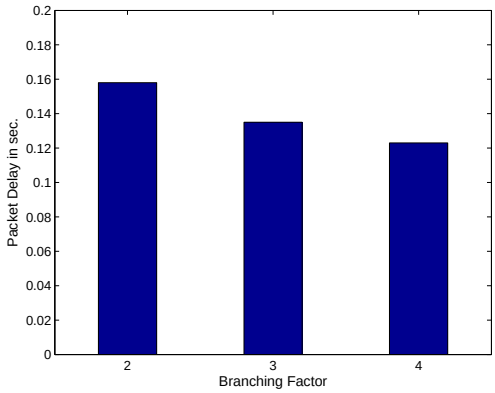


FIGURE 7.2. *Average packet delay for a 21-peer topology*

7.1.3. Peer Join Evaluation

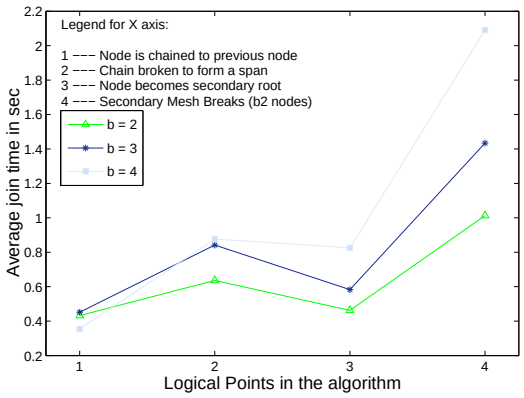
In the next set of experiments, we compare the join times and the mesh management overhead for different topologies with out-going links set to $b = 2$, $b = 3$ and $b = 4$. We measure the join time at a few logical points in the mesh topology.

1. Peer is chained to a previous node. For e.g a new peer is connected to the root of the secondary mesh.

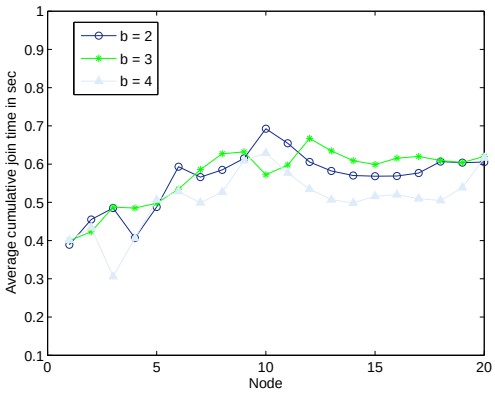
2. Chain is broken to form span. For e.g. a new peer is added to the secondary mesh described in the previous step.

3. Peer becomes secondary mesh root. For e.g. a new peer is connected to the balanced primary mesh

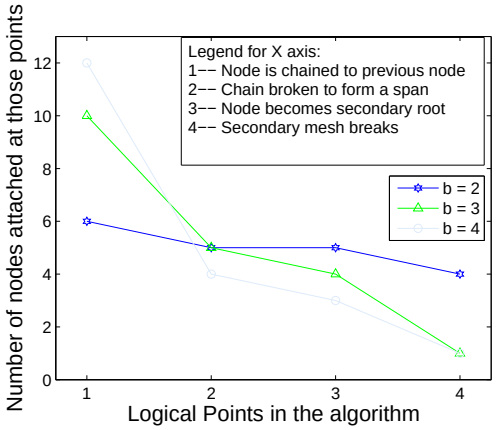
4. Secondary Mesh is destroyed. For e.g. the incoming peer results in peer count in secondary mesh evaluating to b^2 peers. Recall that, in such cases,



(a)



(b)



(c)

FIGURE 7.3. *(a) Average join times at different points in the algorithm for 21-peer topology (b) Cumulative average join time for 21-peer topology (c) Number of peers joining at different logical positions for 21-peer topology*

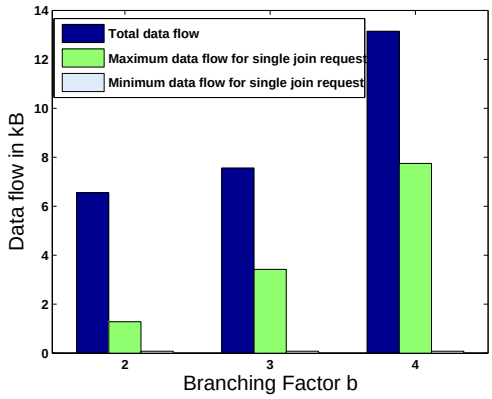


FIGURE 7.4. *Mesh management overhead during join requests for 21-peer topology*

the secondary mesh is destroyed and its peers are reattached to the primary mesh.

Figure 7.3(a) shows the join time for peers at different logical points in the algorithm. As seen, 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, is similar for all topologies. 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 consequently, the join time increases. Figure 7.3(b) shows the cumulative average join times for different values of b. As seen, the average join times for all values of b is approximately the same. Figure 7.3(c) shows the number of peers that join at each logical position which influences the average join times.

Figure 7.4 shows the mesh management overhead at the Supernode in response to the join requests it receives. As seen, the overhead increases with b

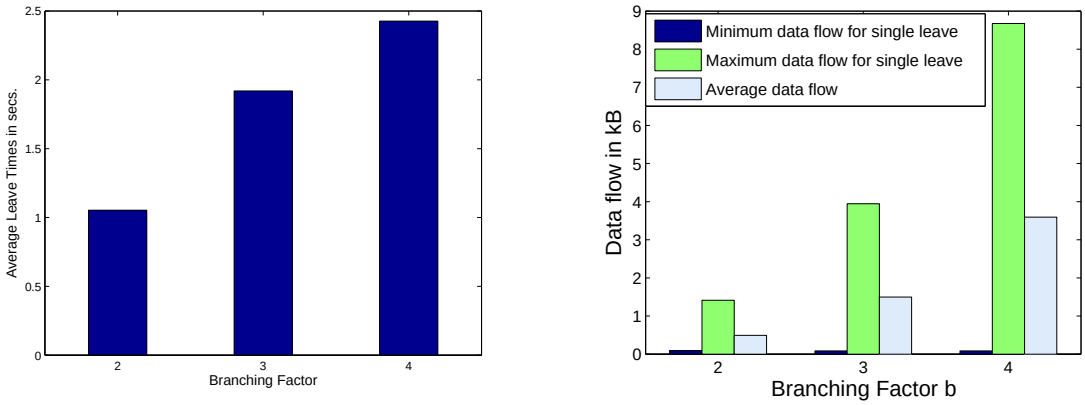
because as b increases, the number of affected peers, to whom messages are sent out, also increases.

7.1.4. Peer Leave Evaluation

Another set of experiments involved measuring the average time taken for the mesh to reconstruct when nodes leave a session. For this experiment, we had in place 21-node topologies for $b = 2$, $b = 3$ and $b = 4$. We measured the leave times for the following scenarios :

1. Secondary Mesh are 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 hierarchy 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.

These scenarios were chosen as observations revealed that these scenarios encompassed the factors that influence leave times the most. Following the recording of these values, averages were calculated for each of the 3 topologies. Figure 7.5 shows the resulting graph.



(a) (b)

FIGURE 7.5. *(a) Average leave times for a 21-peer topology (b) Mesh management overhead during leave requests for a 21-peer topology*

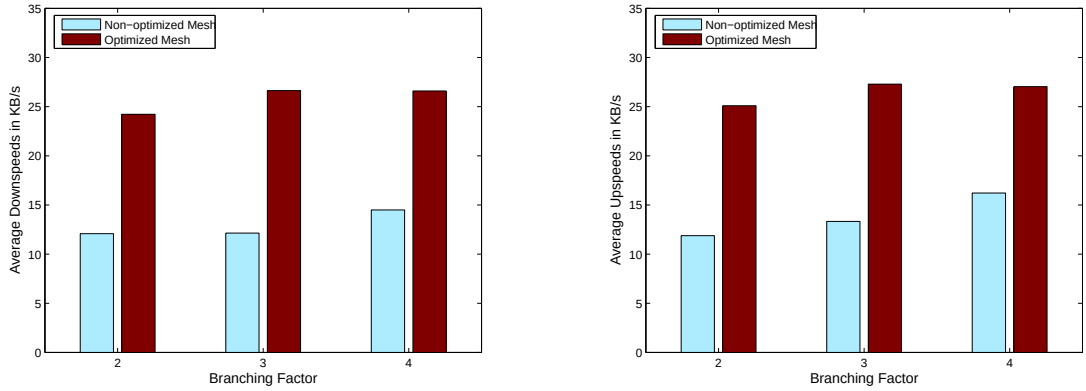
As seen from Figure 7.5(a), with increasing values of b, the leave times also increase. Recall that, the number of nodes affected by a node leaving the session is at most $b^2 + 2b$.

Figure 7.5(b) shows the mesh management overhead at the Supernode when a peer leaves the mesh. Similar to the join overhead, the average overhead increases with an increase in b. Again, the minimum overhead is incurred when a chained peer leaves as it only affects its parent. The maximum overhead occurs when an internal(non leaf) peer leaves a perfectly balanced mesh where, first a leaf peer is swapped with the departing peer and then, a secondary mesh is formed out of the remaining lowermost $b^2 - 1$ peers. As b increases, the number of peers required to form a secondary mesh increases and hence, the overhead increases. Summarizing, on an average, when a peer leaves, the number of peers affected

increases with b and hence, the average overhead in response to leave requests increases with b.

7.1.5. System Throughput Evaluation of an Optimized Mesh

We have implemented an optimized version of the proposed system as described in Section 4.4. For this experiment, we used 15-peer topologies with $b = 2$, $b = 3$ and $b = 4$. The sending rate of the source was set to approximately 30 KBps. The first 3 peers of the mesh were simulated as the peers with very low capacities. Their connection types were set as either DSL (upload capacity = 20 KBps) or Dialup (upload capacity = 10 KBps) whereas the rest of the peers in the mesh were set as T1s (no bottleneck). The results of the experiment are depicted and discussed below.



(a) (b)

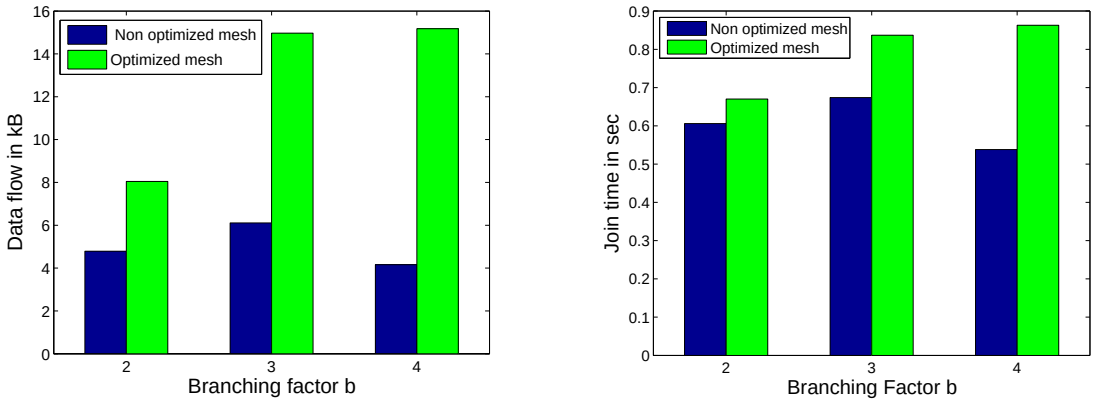
FIGURE 7.6. *(a) Average down-speeds for a 15-peer topology (b) Average up-speeds for a 15-peer topology*

As seen in Figure 7.6, the optimized system performs better than the non-optimized version of the system. This reason can be attributed to the fact that in the optimized system, the weaker upload capacity peers get swapped to the bottom of the mesh and therefore, affect very few other peers. Whereas, in a non-optimized system, if such peers happen to join a session in the early stages, then they end up taking high positions in the mesh and consequently, affect a large number of other peers. Thus, the impact of such low capacity peers in an optimized system is much lesser than that in a non-optimized system.

In the optimized version of the proposed system, when a peer joins in, the Supernode seeks a lower capacity peer higher up in the mesh. If such a candidate peer exists, the Supernode swaps the candidate peer with the new peer. As a result, the messages generated during this join is more than that in the case of a non-optimized system where no swapping occurs thereby, affecting lesser peers during a join. Along with increase in management overhead, the average join time of peers also increases in the optimized system.

Figure 7.7(a) shows the comparison of management overhead for non-optimized and optimized systems for different values of b. The overhead for an optimized system is greater than that for a non-optimized system. Figure 7.7(b) shows the comparison of average join times for non-optimized and optimized meshes for different values of b. As seen, the average join time for peers increases with in the case of an optimized system.

However, we notice that the increase in both the mentioned metrics and therefore, the total overhead introduced is very small. This warrants the use of an optimized system for the advantages it provides.



(a) (b)

FIGURE 7.7. *(a) Mesh management overhead for 15-peer topology (b) Average join time for 15-peer Topology*

7.1.6. Packet Loss Evaluation

Low loss rate is desirable for media dissemination to ensure quality of service. With a P2P architecture, the data transmission rate is more unpredictable since the peers participate in forwarding data packets. Experiments were conducted 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 7.8 shows the change in the loss rate as sending bit rate increases. Two different sessions were executed for each bit rate. The loss rate for a session is the average of loss incurred at all of the receiving peers inside the mesh. The loss rate plotted in Figure 7.8 is the average of these two sessions. As seen, the overall loss rate increases as bit rate increases. However, FEC helps alleviating the loss rate.

In particular, 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%.

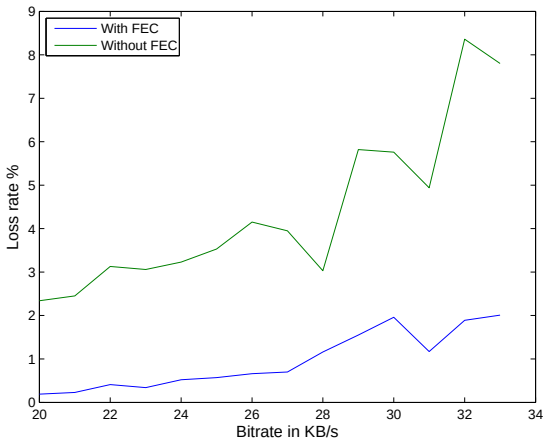


FIGURE 7.8. *Packet loss at different bit-rates*

7.2. Large Scale Simulation

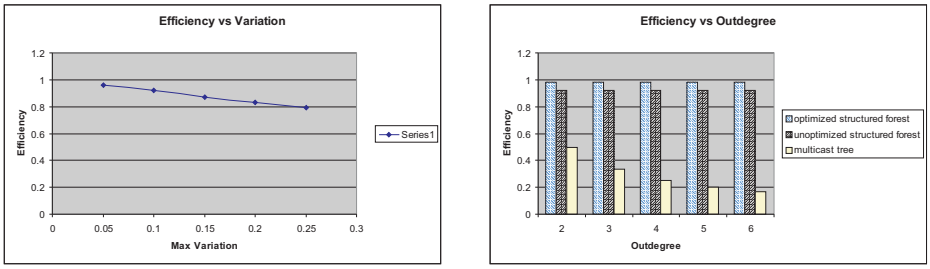
All simulations were done using NS [12]. In our simulations, we used BRITE [13] to generate *Albert-Barabasi* topologies. We present a few results of our simulations in the following sections.

7.2.1. 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 7.9(a) shows the throughput efficiency for our structured mesh vs. maximum variation on capacity v. As seen, an increase in the capacity variation causes a decrease in the throughput efficiency of the system since an internal node with small capacity

may create a bandwidth bottleneck for all the nodes that receive data through it. However, for $v = 0.25$, the throughput efficiency is still 0.8%. Similar results are obtained when node capacity is normally distributed.

Figure 7.9(b) shows the throughput efficiency vs. the out-degree for three different schemes: traditional multicast tree, non-optimized structured mesh and optimized structured mesh. Recall that, for an optimized structured mesh, nodes with lower capacities are moved towards the bottom of the mesh so as to affect a minimal number of other nodes. As seen, throughput efficiency is 98% for optimized structured mesh, 92% for non-optimized 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), that do not contribute to the system throughput, increases in this topology.



(a) (b)

FIGURE 7.9. *(a) Efficiency vs. variation (b) Efficiency vs. out-degree for different data dissemination schemes*

7.2.2. Robustness Evaluation

The following simulations depict the effect of node failure on the proposed topology. An *Albert-Barabasi* topology consisting of 1500 routers was generated

using BRITE [13]. Next, an additional 1000 overlay nodes were randomly generated and connected to the existing 1500 routers. Recall that, two important factors that determine the effect of a node failure of a node on the topology are

1. the position of the failed node (leaf node or internal node), and
2. the branching factor b

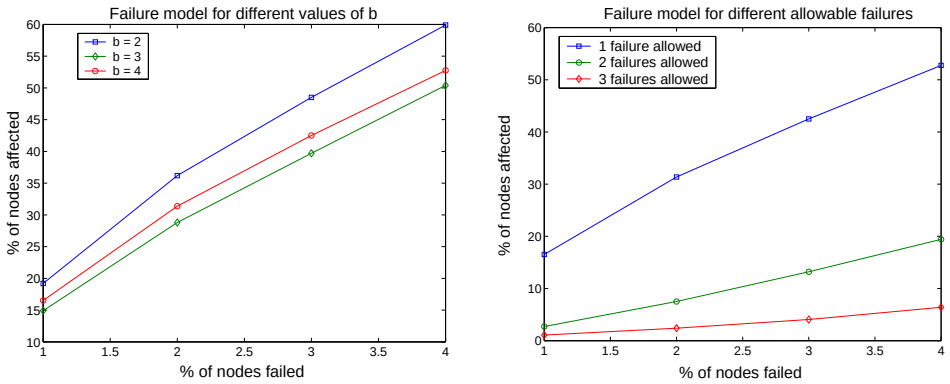
A failed internal node affects more nodes than a failed leaf node since a failed internal node affects

1. its descendants, and
2. nodes in other groups that rely on the affected descendants of the failed internal node for data delivery.

For a failed leaf node, only the nodes in the other groups that receive data from it are affected.

The branching factor determines the number of nodes a particular node is connected to and hence, delivers data to. Hence, the number of nodes affected for a given failed node increases with an increase in the branching factor b.

Figure 7.10(a) shows the percentage of affected nodes as a function of failed nodes for different values of the branching factor. Note that, these failures are temporary as the system detects such failures using the *Heartbeat* feature as described in Section 4.4 and undergoes reconstruction as described in Section 3.6.2. 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. For $b = 3$ and $b = 4$, the number of internal nodes is similar, but due to the branching factor, the affected nodes for $b = 4$ is higher than $b = 3$.



(a) (b)

FIGURE 7.10. *(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 with* $b = 4$

If techniques such as FEC or Multiple Description Coding are employed [14] [15], a node need not receive the complete data. A node is considered a failed node only if it does not receive more than a certain number of partitions K. Figure 7.10(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 since a node receives data from different groups in the topology. Thus, to completely deprive a node of any data, it would require a large number of nodes to fail. For all practical purposes, the probability of such an event is small. Similar performance evaluation has also been carried out in [32].

8. CONCLUSION AND FUTURE WORK

8.1. Conclusion

In this work, we presented a hybrid P2P system, *Hypp*, designed for optimal, synchronous, real-time and non-real-time content distribution from a single source to multiple receivers in a source constraint network. We also discussed the design of the peer subsystem. We discussed the communication and the content delivery mechanisms between peers in the P2P system.

In particular, the contributions of this work include:

1. The notion of throughput efficiency for measuring the throughput optimality of any data dissemination P2P system.

2. a set of optimal P2P topologies designed to achieve high throughput, scalability, low delay, and bandwidth fairness.

3. an implementation of a practical P2P system, *Hypp*, based on the proposed optimal topologies and data dissemination algorithms.

4. discussion of the design of the Peer subsystem of *Hypp* and the communication mechanism amongst Peers to achieve the goal of content distribution.

We have also presented the experimental results of our P2P system consisting of PlanetLab [29] nodes. As observed, the results demonstrate that our approach outperforms traditional overlay multicast tree and achieves near optimal throughput. It also provides high scalability, low delay and bandwidth fairness. In summary, it achieves our goal of simple, efficient, scalable and synchronous content delivery.

8.2. Future Work

We have provided an infrastructure for efficient, scalable and synchronous content distribution from a single source to multiple receivers. Our experimental results prove that the system is very easy to manage and achieves optimal overall throughput. Some absorbing future work is listed below.

1. Functionality could be added so that the system adapts dynamically with changing network conditions. Peers higher in the mesh could be swapped if it is discovered that their performance degrades over time. This could be implemented by having each peer periodically report its statistics to the Supernode.

2. Clustering of nodes could be done so that nodes with similar capacities would lie in the same cluster. Thus, a cluster would consist of a b-unbalanced mesh of nodes with similar capacities. The algorithm could then applied to connect all the clusters together thereby yielding an efficient system in with nodes of differing capacities. (Refer Figure 8.1)

3. Using the content delivery infrastructure provided, exciting applications could be developed. Some of the potential ventures are enumerated below.

   (a) A video streaming application could be built using this content delivery infrastructure. A study needs to be carried out to feed the incoming content to a media player so that the video content could be played almost immediately with some amount of buffering as it is arrives. VideoLan [8] amongst other media players seems to be promising in this respect.

(b) P2P file transfer applications could be developed on top of *Hypp*. *Hypp* could easily be tweaked to use TCP for content delivery as opposed to the UDP data delivery it currently uses.

(c) In case of video streaming applications, work could be done in the field of error correction so that with the employment of FEC or multiple description coding, a better quality of video could be achieved in addition to the efficient delivery content resulting in a better performance.

(d) With a business object layer on top of our content delivery infrastructure, applications such as online classrooms, presentations, video conferencing, etc. could very easily be developed.

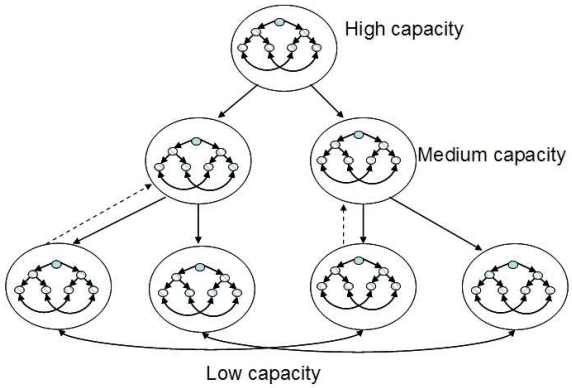


FIGURE 8.1. *Clustering*

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] Jin Li, Philip A. Chou and Cha Zang, "MutualCast: An Efficient Mechanism for One-To-Many Content Distribution," *ACM Sigcomm Asia Workshop*, April 2005.

[3] 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.

[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] http://www.videolan.org.

[9] http://www.edonkey2000.com

[10] http://www.emule-project.net

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

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

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

[14] 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.

[15] T. Nguyen and A. Zakhor, "Multiple Sender Distributed Video Streaming," *IEEE Transactions on Multimedia and Networking*, vol. 6, no. 2, pp. 315–326, April 2004.

[16] T. Nguyen, S. Cheung, and D. Tran, "Efficient P2P Data Dissemination in a Homogeneous Capacity Network using Structured Mesh," in *IEEE International Conference on Multimedia Access Networks*, 2005.

[17] 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.

[18] Y. Minsky, A. Trachtenberg, and R. Zippel, "Set Reconciliation With Nearly Optimal Communication Complexity," in *IEEE International Symposium on Information Theory*, 2001.

[19] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat, "Bullet: High bandwidth Data Dissemination Using An Overlay Mesh," in *SOSP*, October 2003.

[20] 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.

[21] A. Rowstron, A-M. Kermarrec, M. Castro, and P. Druschel, "Scribe: The Design of a Large-scale Event Notification Infrastructure," in *NGC*, November 2001.

[22] 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.

[23] R. Sherwood, R. Braud, and B. Bhattacharjee, "Slurpie: A Cooperative Bulk Data Transfer Protocol," in *IEEE INFOCOM*, 2004.

[24] A. Savrou, D. Rubenstein, and S. Sahu, "A Lightweight, Robust P2P System To Handle Flash Crowds," in *IEEE ICNP*, November 2002.

[25] X. Yang and G. de Veciana, "Service Capacity of Peer-to-Peer Networks," in *IEEE INFOCOM*, 2004.

[26] Z. He, D. Figueiredo, S. Jaiswal, J. Kurose, and D. Towsley, "Modeling Peer-to-Peer File Sharing Systems," in *IEEE INFOCOM*, April 2003.

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

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

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

[30] T. Nguyen, Krishnan Kolazhi, and Rohit Kamath, "Efficient Video Dissemination in Structured Hybrid P2P Networks," in *ICME*, 2006.

[31] T. Nguyen, Krishnan Kolazhi, and Rohit Kamath, "Efficient Content Distribution on Source Constraint Networks," in *IEEE Transactions on Multimedia*, 2006.

[32] Krishnan Kolazhi, "Node and Topology Management for Content Distribution in Source Constraint Networks," *MS Thesis Report, Oregon State University*, 2006.

APPENDICES

APPENDIX A. Proofs

The proofs discussed in this section are similar to those in [31].

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

Proof :

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. Since the total sending rate $E = \sum_{i=0}^{i=N} S_i$ and the total receiving rate must equal to each other, and therefore 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 holds true.

(a) The throughput efficiency for this scheme $E = 1$.

(b) Node delay is constant.

(c) Node insertion and deletion affects at most $O(N)$ nodes where N is the number of destination nodes.

(d) The out-degree of any node in this scheme is $O(N)$ where N is the number of destination nodes.

Proof :

(a) Each destination node receives C/N bps from the source node and broadcasts

this 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 $N(N-1)C/N+C = NC$ bps. In this scenario, $\min(\sum_{i=0}^{i=N} C_i, NC_0) = NC$ and hence $E = 1$.

(b) By construction, each destination host is at most 2 hops away from the source node and therefore the delay is constant.

(c) Since, by construction, each destination node is connected to $N-1$ other destination nodes, a node joining or leaving the topology will affect all other nodes and hence is $O(N)$.

(d) By construction, each node connects to $N-1$ other destination nodes in the topology. Therefore, the out-degree of a node is $O(N)$.

Theorem 3 : For a *Chain Topology*, the following properties are satisfied.

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

(b) Node delay is $O(N)$ where N is the number of destination nodes.

(c) Node insertion and deletion affects a constant number of nodes q.

(d) Out-degree of a node is constant.

Proof :

(a) Each destination node, except the last node in the chain, receives C bps and broadcasts its data to 1 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 which equals to the total receiving rate of all the nodes and hence $E = 1$.

(b) Since it takes N hops for a data packet to reach the last destination node in the chain from the source, the delay for this scheme is extremely large i.e. $O(N)$.

(c) Since any node is connected to at most 2 other nodes in this scheme, node insertion or deletion affects a constant number of nodes.

(d) By construction, any node sends data to at most one other destination node and hence, the out-degree of a node is constant.

Theorem 4 : For a *Balanced Mesh Topology*, the following hold true.

(a) The throughput efficiency $E = 1$.

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

(c) The out-degree of any node is at most b.

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 relay 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 the 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) 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, there are $i = log((b-1)N + b) - 1$ hops from the source 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 $log((b-1)N + b) + 1$.

(c) 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 $b-1$.

Theorem 5 : For a *Cascaded Balanced Mesh Topology*,

(a) The throughput efficiency, $E = 1$.

(b) The delay for the cascaded b-balanced mesh is $O((log_b N)^2)$.

(c) the out-degree of any node is at most b.

Proof :

(a) This holds true since each cascaded mesh is a b-balanced mesh where the root of the secondary mesh 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 at most i such meshes and each mesh has depth of $O(i)$, the total delay is therefore $O(i^2)$ or equivalently $O((log_b N)^2)$. Specifically, 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* topology.

Theorem 6 : For a *b-Unbalanced Mesh Topology*, the following properties hold:

(a) Throughput efficiency for this scheme, $E = 1$.

(b) Node delay is $O(\log_b N) + c$ where c is a constant.

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

(d) Out-degree of any node is at most b.

Proof :

(a) For a b-unbalanced mesh, a secondary mesh is constructed according to the algorithm for cascaded balanced mesh. We already know that, for a cascaded balanced mesh topology, $E = 1$. When the secondary mesh is broken, it's nodes are attached to the primary mesh. After the reconstruction, there still remain only b rightmost nodes in b groups each having C/b unused bandwidth. Thus, the total unused bandwidth in the system is C. Hence, similar to a balanced mesh, the efficiency in this case too is 1.

(b) The delay of the root node in the first secondary mesh is $\lfloor \log_b(N+1) \rfloor + 1$ as the root node of the first secondary mesh receives b different partitions from each of the b 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, and one more hop to the secondary mesh's root node. Now, the secondary meshes consist of many balanced meshes cascaded together as described in Section 3.5. Each of these balanced meshes 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 of the root of the first secondary mesh. 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 less than $3b-5$ hops and the throughput efficiency still equals to 1.

(c) The largest 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 to or detached from during the construction or destruction of the secondary meshes. Furthermore, there are also b ancestors, one from each group that need to receive data from the new b nodes (e.g. node 3 in Figure 3.6(b). Unlike the delay of $O((\log_b N)^2)$ for the cascaded balanced mesh topology, the delay for the unbalanced mesh topology is only $O(\log_b N)+c$ where c is a constant.

(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. Thus, out-degree of any node is at most b.

APPENDIX B. Pseudocode

Algorithm APPENDIX B.1: BALANCEDMESH(N)

Construct balanced tree with source as the root

and each internal node with out-degree as b.

for $i \leftarrow 0$ **to** $b-2$

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

for $i \leftarrow 0$ **to** $b-1$

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

Algorithm APPENDIX B.2: CASCADEDBALANCEDMESH(N)

while $N <> 0$

do
- Construct a b-balanced mesh of such that depth
- $i \leftarrow \lfloor log((b-1)N+b) \rfloor - 1$.
- **comment:** The above statement will create the deepest
- **comment:** b-balanced mesh without exceeding N
- **if** exists previous b-balanced mesh
  - **then** Connect b rightmost nodes to root of new balanced mesh.
- $N \leftarrow N - (b^{i+1}-1)/(b-1)$

Algorithm APPENDIX B.3: JoinUnbalancedMesh(i)

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

then
- Set new node as the 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 the b balanced mesh algorithm.
- $sec\_mesh\_node\_count \leftarrow sec\_mesh\_node\_count + 1$
- **return** (0)

if $sec\_mesh\_node\_count == b^2$

then
- **for** $i \leftarrow 0$ **to** $b - 1$
  - **do** Connect b sec. nodes to leftmost node of lesser depth in group i.
- **for each** leftmost node P of a group
  - **do**
    - Disconnect P connections to b-1 nodes of other b-1 groups.
    - Disconnect P connections back to its parent.
- **for each** group of newly attached b nodes
  - **do**
    - Establish their cross links with other groups as described in the Section 3.4.
    - Connect all but the rightmost node to their parent P.
    - Connect rightmost node to highest ancestor without in-degree b
- $sec\_mesh\_node\_count \leftarrow 0$
- **return** (0)

Algorithm APPENDIX B.4: LeaveUnbalancedMesh(i)

if node is in primary mesh

then
- **if** secondary mesh exists
  - **then**
    - Swap leaving node with a node in the 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.
      - **comment:** These $b^2 - 1$ nodes are the siblings of the replacement node
    - **else**
      - Construct a secondary mesh with remaining $b^2 - 1$ nodes.
      - **comment:** No need for swapping in the above case

else
- Reconstruct the secondary mesh with on one lesser node.
- **comment:** node is in secondary mesh.