Implemention and Evaluation of PRIME, a Peer-to-Peer Streaming Mechanism for Live Video

Jimmy Hastings

August 2009

Abstract

User-generated content is becoming increasely popular on the Internet, and peer-to-peer networks provide opportunity for scalable delivery thereof. However, the ability to transmit video to a large number of people remains in the hands of those with significant resources, who become de facto gatekeepers. I examine one protocol, PRIME, designed to bridge this gap by providing live, peer-to-peer streaming. I implement the protocol for the first time and test it for desired properties, providing convincing evidence that this system could be deployed in real-world situations.

Acknowledgements

I would like to thank my committee: Professors Young, Rosenow, and especially Professor Rejaie for his guidance throughout the project. I would also like to thank Professor Cheng for her advising assistance. I would like to thank the UO CIS systems staff for allowing me to use and assisting me with the use of the local machines. Finally, I would like to thank Nazanin Magharei, without whose assistance this project would be a fraction of what it is.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Recently, the Internet has seen a boom in user-generated content, including pictures, podcasts, video, and any other type of file.. Anybody who so desires can produce and share something interesting with whomever they please, be they videos on YouTube, podcasts on iTunes, blogs on Blogger, or any number of other media and services. Unlike with television, there is now no clear distinction between those who provide content for others to experience and those who consume the content. This allows a much richer variety than anyone had previously thought possible.

Unfortunately, the resources required to transfer video to large groups of people are still large enough that the ability to do so remains in the hands of a few de facto gatekeepers, including television networks and Internet companies with large server farms. Despite coming from a myriad of users, YouTube videos are ultimately controlled and transmitted by Google, a company with the resources and the willingness to do so. For a sporting event, if the right person or company does not take an interest and commit their resources, that game will not be broadcasted.

The tradditional approach for these applications is client-server, requiring the content provider to have bandwidth resources proportional to the number of users. However, peer-to-peer technologies scale with the number of users, and can therefore bridge this resource gap for some applications. BitTorrent [1], for example, allows someone to share a file with an unbounded number of users, while only requiring enough bandwidth to share one copy of the file, rather than either requiring enough bandwidth to send to each recipient, or relying on a third party such as RapidShare [2] to make up the bandwidth difference.

This is because peer-to-peer networks can utilize the bandwidth of every user in the system. Because each peer can receive the data in any order, different peers can start with different parts of the file, and share it with each other until everybody has the entire file. This works for static files, such as computer programs or individual songs, but fails for streaming content, such as YouTube videos or television, because it matters in what order the data is sent if it is being streamed.

1.2 Motivation

The ability to stream high-quality media to a large number of people need not be limited to those who can afford the bandwidth. If we can bridge that gap, we can see an even more diverse, vibrant Internet. Any sports team, no matter how small or large, could broadcast their games to any size audience, no matter how large or small. A grandmother at her home would be able to produce cooking shows for hundreds of viewers with only a residential Internet connection. A small group of people could easily start an Internet radio or tv station dedicated to some niche group or interest. This kind of technology would even have implications for large corporations, in that they would no longer *need* to invest in all the infrastructure required to send the same

data to thousands of different people.

However, even though we have technologies such as BitTorrent for static files, a similar solution for streaming content is not yet widely available in the real world. This is because peer-to-peer streaming offers challenges unique from other contexts. As in all peer-to-peer services, we must be concerned with effectively utilizing each peer's contributed bandwidth, including deciding who connects to whom (this is called the overlay). Unlike with static files, however, we also must ensure that each peer receives their required data in a timely fashion. Because each peer is constantly consuming data, one cannot simply send the data in a random order, as BitTorrent does. Timeliness is especially important when dealing with live content instead of a video on demand setting, as users will expect very little delay if content is, for example, a New Year's Eve countdown or a sports game. However, we can also use the knowledge that each user is transmitting content at the same time to our advantage.

There are a few protocols that attempt to achieve the goal of peer-to-peer streaming of live content, such as CoopNet [3] and PRIME [4]. Each of these systems addresses the previous concerns in slightly different ways, but none of these are yet deployed for mass-market consumption.

1.3 Main Focus

The focus of this thesis is on implementation and evaluation of PRIME, the peer-to-peer receiver-driven mesh-based streaming protocol developed at the University of Oregon. While this protocol had been previously tested using simulations [5], I have implemented this protocol for the first time to conduct real-world experiments on both a local testbed of computers, and the Internet.

Using this code, I ran a series of experiments to test how well PRIME performs, first over a small set of local machines, then over PlanetLab [6], a worldwide testbed of networked computers. Analyzing the output from these machines, we show efficiently each peer uses its bandwidth, and how quickly packets are able to diffuse throughout the nodes. We also show how an increase in source bandwidth affects the performance of the system.

1.3.1 Findings

My main findings from experiments are as follows:

- Over a local testbed, PRIME is able to efficiently use resources to deliver a high quality of content to all users
- Globally, PRIME is suitable for large, distributed deployment, with inefficiencies only occurring on unusually high-bandwidth connections.
- Bandwidth utilization spikes downward immediately after the startup phase has ended on local tests. This is not significant for final efficiency, but should be resolved in the future.

1.4 Road Map

The rest of this report is organized as follows: In chapter 2, I describe the challenges and benefits of peer-to-peer networks, specifically focusing on the differences between dealing with static files and streaming content. In chapter 3, I examine PRIME in-depth, specifically focusing on the details as they relate to my implementation and evaluation. Chapter 4 describes my implementation, both of the PRIME protocol and the analysis scripts I needed to consolidate the information contained in the

various log files. I pay specific attention to my major design decisions with respect to architecting the code for maintainability and ease of debugging. I then discuss in chapter 5 the performance evaluation metrics of a peer-to-peer streaming system, as well as the practical concerns surrounding data collection and diagnosis of problems. Finally, in chapter 6 I will discuss my results, both from experiments on the local testbed and worldwide. I discuss what this means for PRIME and for the Internet, and what further work in this area is needed.

Chapter 2

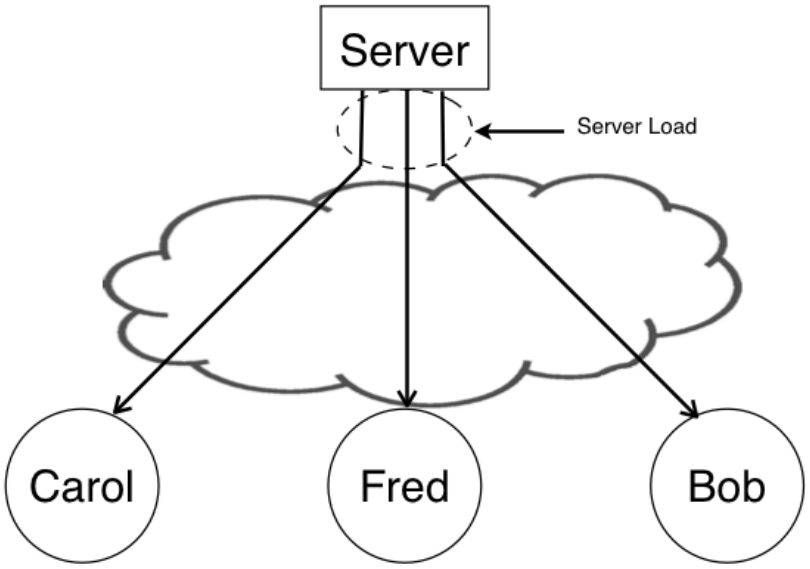
The Advantages and Challenges of Peer-to-Peer Systems

This chapter provides context for this study by first, in section 2.1, explaining the main idea behind peer-to-peer systems, then explaining how they differ from a traditional client-server model, and why they are more scalable. Section 2.2 then discusses the difference between static and streaming content, and the challenges specific to this study's focus of peer-to-peer live streaming. Section 2.4 examines two ways in which one can generate an overlay for a peer-to-peer system, and the advantages and disadvantages of each.

2.1 Architectures

The ways of distributing content can be roughly divided into two categories based on the interactions between sender and receiver.

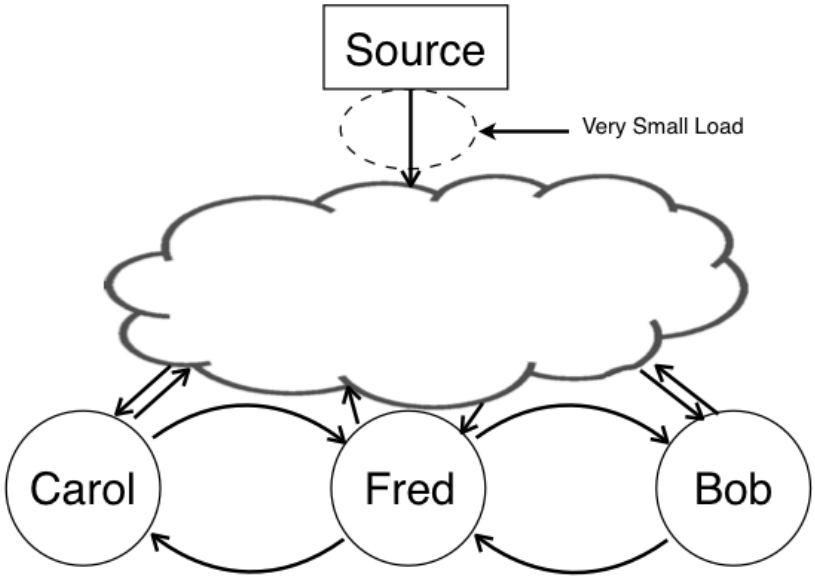
Figure 2.1: Traditional client-server model. Note that all connections come from the server.



2.1.1 Client-Server Architecture

The traditional model is a client-server approach, as shown in figure 2.1. One entity, the server, has all the content, and each client that wants to receive the content should directly connect to the server. The advantage to this scheme is that it is easy to set up and easy to control. As long as the server is up and has the required resources (mainly bandwidth), it can provide the content. The main problem with this approach is that it is not easily scalable. If the server is streaming content that requires x kbps delivery rate to n clients, the server needs $n * x$ kbps of bandwidth to deliver packets to all clients in time. That is, twice as many clients means twice as many resources that the server needs.

Figure 2.2: Peer-to-peer model. Peers mainly share data with each other.



2.1.2 Peer-to-Peer Architecture

The main idea in a peer-to-peer architecture is that each user who receives content also helps to send it. In essence, each client now acts as a mini-server. The main advantage of this approach is scalability. That is, if there are twice as many peers in the system, we automatically have twice as much bandwidth to deliver it. Both demand for resources and available resources scale with the number of users, whereas in a client-server model, only demand scales.

There are already several protocols in use that use a peer-to-peer architecture for sharing some types of content. Companies use peer-to-peer protocols to distribute video game demos and large video game updates that will be in high demand without putting a strain on their servers. Likewise, artists might make their content available through peer-to-peer file-sharing servers to lessen the burden on their servers. Criminals, too, use certain peer-to-peer systems to share copyrighted content. All these applications use peer-to-peer architectures to distribute files.

Unfortunately, once we begin to implement a peer-to-peer architecture, our models become much more complicated. Peer-to-peer systems must meet several challenges to be effective, including how peers decide what data to transfer, how peers find other peers with which to connect, and how to deal with some peers having higher bandwidth than others. These challenges are examined in more detail in section 2.3.

2.2 Classes of data

Content can be roughly divided into two classes, based on its delivery style.

2.2.1 Static Content

Most of the content sent over the Internet consists of files that are considered static in nature. That is, a user will request a file that has been created at some point in the past, and the data does not change over the course of the data transfer. The user then consumes the data after it has been completely transmitted. When a user views a web page, he is requesting a static file, no matter how often the file changes on the server, or if it was generated specifically for the user. The user cannot view the content until it is finished transferring. Likewise, many peer-to-peer systems such as BitTorrent deal in the realm of static files, a user does not require the first part of the file before the last part— he needs the entire file to consume it anyway.

2.2.2 Streaming Content

Content is considered to be streaming if the client consumes it as the content arrives. For example, audio or video content is usually streaming, and they often have a natural rate for delivery based on how much data is consumed each second. Streaming

content can be further divided into two types based on when the content is intended to be played.

Stored Streaming Content

In this situation, content is created and stored on a server. When a user wishes to view this content, he requests it from the server and it is transferred continuously. The user begins to consume it as soon as he has enough to consume, while the data is still being transferred. YouTube is perhaps the most visible example of a service for stored streaing content. Comcast offers a video on demand service through their cable television that works the same way.

Live Streaming Content

On the other hand, live content is being created as it is being transmitted. This means that each user that consumes the data does so at roughly the same time. Examples include live Internet television and radio programs. Sports events are a prime candidate for live transmission, because fans usually want to see the scores and results as they happen.

2.3 Challenges in Peer-to-Peer Live Streaming

PRIME operates in the realm of peer-to-peer streaming of live content, and must address the following challenges associated with it.

First, as with all peer-to-peer systems, we must be concerned with effectively utilizing the bandwidth of all clients to achieve scalability, when initially only the source has content to send to anyone. We must decide how to connect peers to each other, and how to distribute content on those connections.

BitTorrent [1] solves this problem by forming a random mesh, where each peer connect to a random subset of peers. Initially, each peer is sent a different portion of the file, so they are likely to have different pieces than their neighbors. This works because each peer cannot consume the data until it has finished transferring. However, in a streaming context, we must be concerned with the order of arrival. If we are to use a swarming method as BitTorrent does, how do we properly incorporate new data into the mesh?

Other systems, such as CoolStreaming [7] and ChunkySpread [8] form multiple trees, sending a different portion of the content through each tree. Ultimately, this approach suffer when peers enter and leave the system while streaming [9].

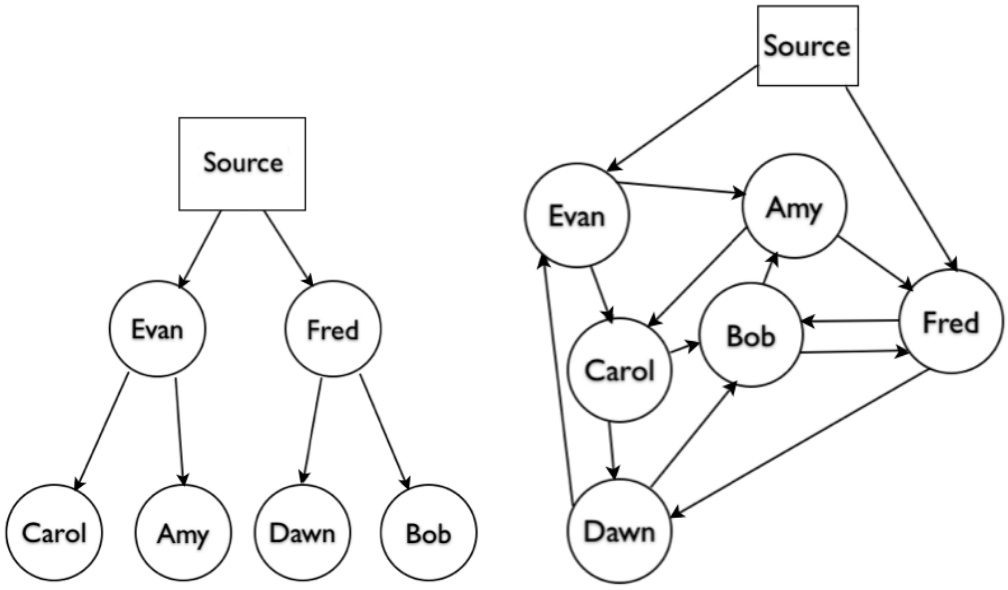
Once the overall behavior of the system is determined, we must decide how each agent in the system acts to achieve the goal. Peers are often working with incomplete information about the nature of the system when making individual decisions.

2.4 Overlay Structure

Different peer-to-peer systems have solved the problem of creating an overlay in different ways. These can be categorized as one of two basic overlay structures: tree and mesh.

The more structured way is with a tree rooted at the source, as seen in figure 2.4(a). That is, each node will have one parent and a set number of children (the degree of the tree). This allows the system to push the content in an orderly fashion, and guarantee that all peers get what they need within a reasonable amount of time (the delay is proportional to the depth of the tree). The problem here from an efficiency standpoint is that the nodes at the bottom (the leaves of the tree) do not use their outgoing bandwidth to help the system. Because at least half of tree nodes

Figure 2.3: Two structures for peer-to-peer overlays



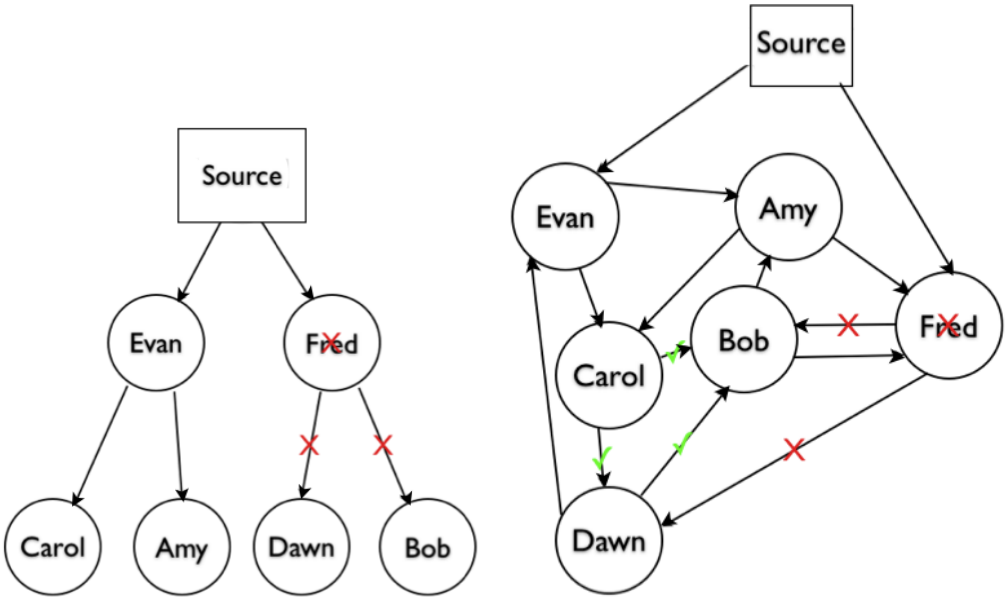
(a) Tree-based approach (b) Mesh-based approach

are leaves, over half of the potential system bandwidth is left untapped. There is also an issue with durability. If one of the nodes at a high level leave, either due to network outages or by choice, none of their children can get that content, and the whole subtree withers, as seen in figure 2.5(a). One needs trusted internal nodes for this scheme to work well.

CoopNet [3] handles this by forming multiple trees, and having different channels pushed down different trees. The trees are built such that each peer is an internal node for one of those trees. This solves the efficiency problem, but magnifies the problem of peers departing.

An alternate method is to have each node connect to several random nodes in the system. This is known as a mesh (Figure 2.4(b)), and is the scheme used by BitTorrent [1]. Now, each node has several parents, so if one parent leaves, as seen in figure 2.5(b), peers are not left without data entirely. Additionally, peers that

Figure 2.4: What if Fred leaves?



(a) Tree-based approach: Dawn and Bob get no packets.

(b) Mesh-based approach: Carol can still provide packets to Dawn and Bob until they get an additional parent.

lose a parent will be able to request a new connection without forcing the overlay to restructure itself. New nodes are likewise seamlessly integrated into the system in the same way, because position in the mesh is assigned randomly.

The main issue with a random overlay is that the system loses the ability to orderly distribute content. Because the system is random, it is difficult to guarantee that every peer will receive timely content. Likewise, it is difficult to guarantee that each node can efficiently use its outgoing bandwidth.

Chapter 3

The PRIME Protocol

The main idea behind the PRIME Protocol [4] is that peers form a random mesh and use swarming content delivery on top of the mesh. This is similar to what BitTorrent [1] does, but with some important distinctions unique to the problem of live streaming content. This chapter describes the two main components of the PRIME protocol:

- Overlay construction (section 3.1.1)
- Content delivery (section 3.1.3)

Section 3.2 examines how the actions of each node affect the overall goals of the system.

3.1 Main Components

3.1.1 Overlay construction

One of the major components of any peer-to-peer system is method of overlay construction [9]. In PRIME, there is one node, called the bootstrap node, that keeps track of all the peers in the system. When a peer wants to join the system, it sends

a UDP packet to the bootstrap node, informing it of the peer's presence and it how many parents are needed. The bootstrap node, after a short delay to give other nodes a chance to enter, then sends a random subset of the participating peers to the requesting peer as potential parents. If at any point a peer needs additional parents, it sends a new request to the bootstrap node. Also, periodically each peer sends a heartbeat to the bootstrap node to indicate that the peer is still in the mesh. This makes the system more robust in the case of peers being disconnected.

Key to the PRIME protocol is the organized view of the peer mesh. We define the distance of a peer p from the source as the shortest path from the source to p. Peers can now be grouped based on their distance from the source into levels. If each peer has an out degree of d, and level n has $p(n)$ nodes, $p(n+1) <= p(n) * d$.

One can visualize this by placing peers of each level in the same row of the graph. The mesh from figure 2.4(b) on page 12, when viewed in this way, looks something like figure 3.1.

From this, we can see that there are two kinds of edges: straight edges that go from a higher level to a lower level, and curved edges that don't. Parents that are in a higher level than the child are called diffusion parents. Parents on the same or lower level are known as swarming parents. This distinction becomes very important when delivering content.

3.1.2 Video Encoding

The video that PRIME transmits is encoded by a method method where data is split into several layers (or descriptions), and further split into *e.g.* 50-millisecond packets. As long as a client has one description per packet slice, it can play out the movie, but the more descriptions it has, the higher quality the video that results. Note that the

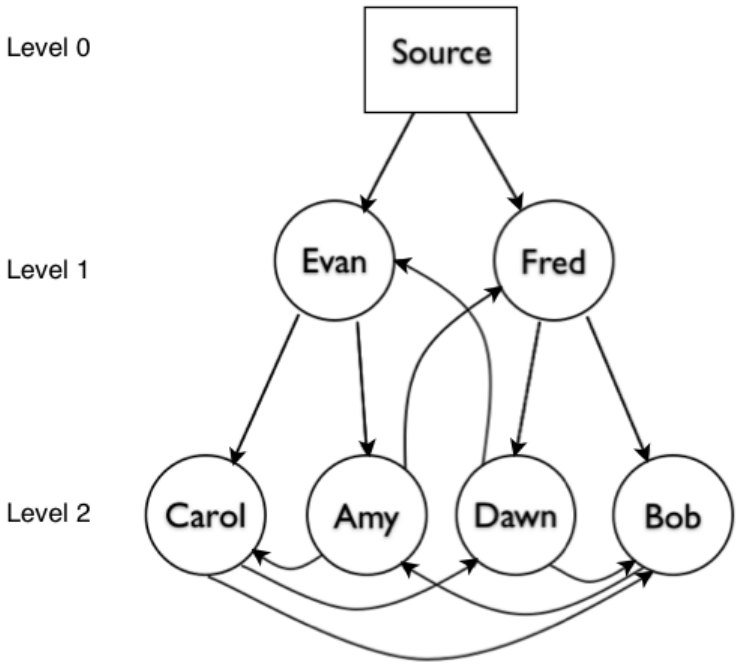


Figure 3.1: Organized view of mesh

peer need not have the same description for each packet slice. For example, it might have description 2 for timestamp 50, and description 5 for timestamp 100, and that is as good as having description 4 for both timestamps 50 and 100.

Each packet is a standard size, and a combination of packet size, packet duration, and maximum number of layers determines the bandwidth required to transmit or receive the video. This is known as the stream rate. For example, if each packet is 5120 bits, lasts 50 ms, and there are 8 potential descriptions, the stream rate is $5120 * 8/50 = 819.2$ bits/ms, or 800 kbps. The source's bandwidth needs to be at least the stream rate for all the packets to get out into the overlay.

3.1.3 Swarming Content Delivery

As mentioned in section 2.3, our main goal with a peer-to-peer system is to effectively use the bandwidth of each participating peer. This means that each parent needs to be able to send something useful to its children. In a streaming context, we have the additional requirement of ensuring that each packet is delivered to peers before its playout time.

With those goals and requirements in mind, PRIME dedicates diffusion edges to delivering new packets. This helps to ensure proper diffusion of new packets throughout the mesh. This allows the swarming parents to have sufficient variety to swarm the remaining packets.

3.2 Packet Scheduling

PRIME uses a receiver-driven, or "pull"-based content delivery system. To that end, as each peer receives packets, it reports the new packets it has received to its children, piggybacked in the header of a content packet. Every Δ seconds, each peer examines the packets its parents possess, and decides which packets to request from each parent, and in what order. If a parent no longer has any requests from its child, it will send marked packets until it receives more requests. This serves as a signal, both to inform the child that it should request more packets, and to allow the child to accurately know the bandwidth of its parent for future scheduling operations.

Each child requests the number of packets it thinks its parent can provide, based on past received bandwidth from that parent. Using an exponentially-weighted moving average (EWMA) of past bandwidths, a child requests the estimated number of packets the parent could provide in this scheduling period, plus 2 standard deviations.

Peers maintain a buffer of ω seconds of data. This should be approximately

$(h+3)*\Delta$ where h is the maximum expected level to contain peers, or $(log_d(n)+3)*\Delta$ where n is the expected size of the mesh, and d is the out-degree of peers (the number of children each peer is expected to have). Because new packets are given priority, we expect each new packet to reach the next level in Δ seconds, the time between scheduling periods. Therefore, we expect packets to have permeated the mesh in $h*\Delta$ seconds. We allow $3 * \Delta$ seconds for the packets to swarm through the mesh. Note that ω must be the same for every peer. We cannot have certain peers decide to keep more or less of a buffer because of personal bandwidth.

3.2.1 Buffer Windows

When scheduling, peers organize the buffer into three different windows, based on relative timestamp. Peers select packets to request based on which window they belong.

Diffusion (New) Window

This window is the highest priority, and it consists of the newest packets the peer knows about. This is the range $[t_{max} - \Delta, t_{max}]$ where t_{max} is the timestamp of the latest packet that any parent has reported as available. This window ensures that new packets permeate the mesh quickly. We can identify diffusion parents by the ones that possess packets in the diffusion window.

Playing Window

This window is of second highest priority, and it contains the packets that we need to play out before the next scheduling event. This is the range $[t_p, t_p + \Delta]$ where t_p is the lowest timestamp in the buffer. Every Δ seconds, we slide the buffer forward,

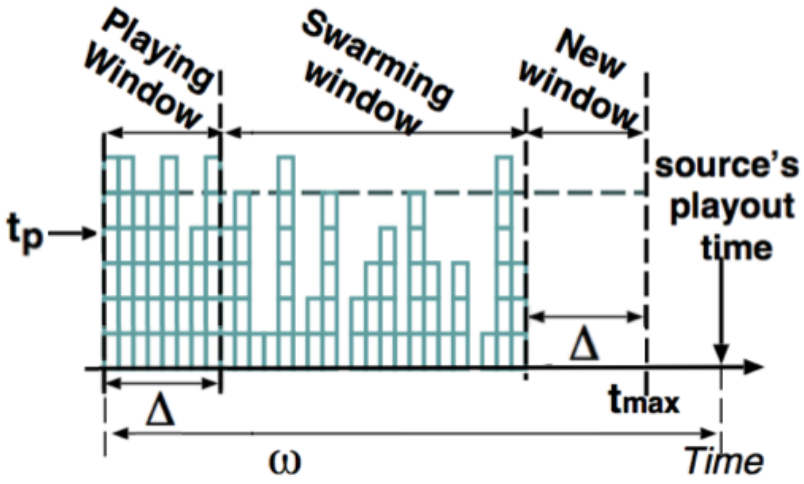


Figure 3.2: Buffer maintained by each peer, with the three windows labeled

advancing the playback window by Δ.

Swarming Window

These are the packets that are currently being swarmed in the mesh. These packets have the lowest priority, but this will be the largest window. This is the range $(t_p + \Delta, t_{max} - \Delta)$ with t_p and t_{max} defined as before.

3.2.2 Packet Assignment

Once a packet has been chosen from the appropriate window, it is assigned to the request list of a parent. When multiple parents could provide the packet, a child will request it from the parent whose request list is least full at the point of assignment, with capacity decided by the EWMA of received bandwidth from that parent.

3.2.3 Quality Adaptation

Each peer has an expected quality. This is the number of descriptions per packet slice it is attempting to consume. Each scheduling operation consists of two rounds. In the first round, a peer will not request a packet that would give him a higher quality than it conservatively expects. After that, if it appears that the parents have any bandwidth left, a child will request any extra packets.

Every Δ seconds, before requesting new packets, each peer goes through a process called quality adaptation (QA). This uses the EWMA of useful packets the peer has received, along with the current state of the buffer, to decide if the peer needs to add or drop a description. A peer will drop a description if the number of packets he expects to receive in Δ seconds is less than the number it would need to fill Δ seconds of its swarming buffer to the target quality. A peer will add a description if the number of packets it expects to receive in Δ seconds minus one standard deviation exceeds what would be required to fill Δ seconds of his swarming buffer to one greater than the target quality.

3.2.4 Startup Phase

When a peer first joins an overlay, it has no packets to offer or to play. Before the node can operate as a normal peer, it goes through a startup phase which lasts $\omega/2$ seconds. During this period, the goal is to be able to play as quickly as possible. To that end, a peer will never attempt more than one description. It will also perform a scheduling event whenever it receives a marked packet from a parent who has previously sent something useful. This speeds up the rate at which new packets are distributed when most of the peers have no packets.

Because startup phase only lasts $\omega/2$ seconds, and peers begin playing the content

after startup phase, it cannot start until the source has at least $\omega - {}^{\omega}/_{2} = {}^{\omega}/_{2}$ seconds of data. If there are peers in the mesh when the source reaches ${}^{\omega}/_{2}$ seconds, all peers are notified to begin their startup phase.

3.2.5 Passive Coordination

Because the source node is the only one that starts out with all the packets, its bandwidth is particularly precious. That is, if the source only has enough bandwidth to send every packet once, then sending one duplicate packet means not delivering a different packet into the overlay at all. Therefore, when a child requests a duplicate packet from the source, it will instead look in $[r, min(r + 2 * \Delta, t_{max})]$, where r is the requested timestamp, and send the least sent packet instead. In the case of ties, the earliest packet is chosen to send.

This is known as passive coordination, because it subtly coordinates the peers at level 1 to ensure that the mesh gets all the packets that were generated. Because diffusion packets are of highest priority, and the source has every packet, level 1 peers will never request playing window packets, so we can be certain this behavior does not prevent a child from playing. Even when the source has excess bandwidth, passive coordination is beneficial because it ensures an even mix of all the packets in the overlay, and can send newly-created packets even before its children's next scheduling events.

Chapter 4

Architecture of Implementation

Although PRIME had been tested in ns [10] simulation, there had not been any implementation of the code to run on live computers over the Internet. This study involved an implementation of the PRIME protocol in C++ to work on both Apple i386 iMacs and Fedora i686 environments, as well as support scripts to run experiments over local machines and PlanetLab.

This chapter details the architecture of this PRIME implementation, as well as the rationale behind some of the design decisions. It notes both the overall structure and some of the smaller details that may not be obvious from the PRIME specification. Finally, section 4.4 touches on the support code that needed to be written to assist with running the experiments and analyzing the data.

4.1 Goals

When writing any piece of software intended for long-term use, a main goal is maintainability, both for software writer and those that may view it later. To that end, this code implements the standard practices of modularity, organization, and docu-

mentation.

This implementation is split into two main levels, one of which implements basic network functionality and houses the main program loop, and the other implements PRIME-specific and role-specific functions. These two levels can communicate in a flexible, robust way that allows the upper level to be programmed in an event-driven manner. Figure 4.1 shows an example of such communication.

4.2 Lower Level

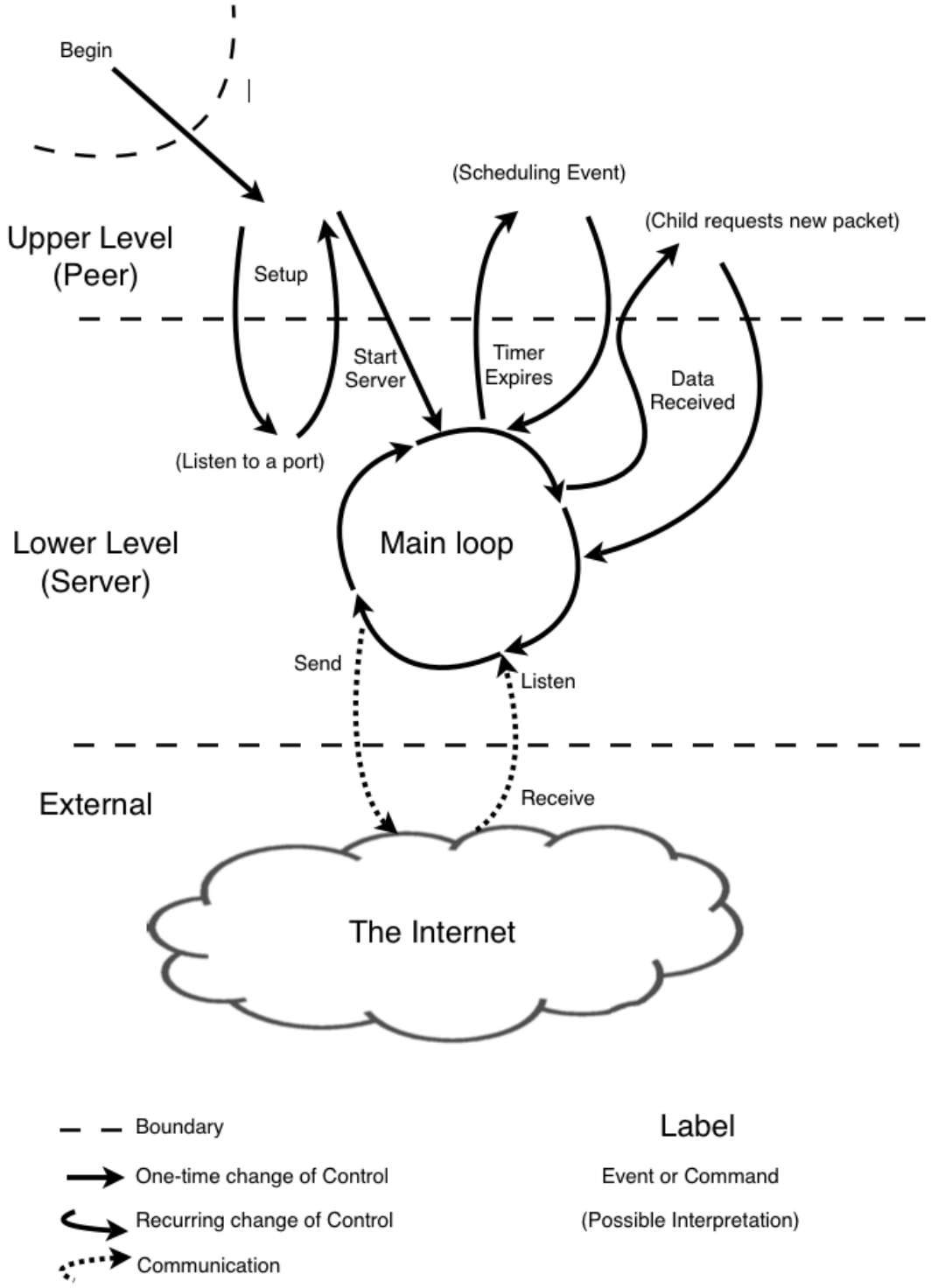
The lower level, stored in `server.cpp`, handles interacting with the TCP and UDP sockets, and notifying the upper level if certain events happen, *e.g.* a timer has expired, or data arrives from the network. This is also what houses the main loop body of the program, and is the same for source, peer, or bootstrap nodes. The main interface functions of `server.cpp` are as follows:

- `listenTCP(port,onConnect,onData,onClose)` causes the server to listen on the specified `port`, and to call the function `onConnect()` when a peer connects, `onData()` when data is received on a connection generated from this listening on this port, and `onClose()` when said peer disconnects.

- `connectTCP(ipaddr,port,onConnect,onData,onClose)` connects the server to a specific IP address (`ipaddr`) at the specified `port`, notifying the upper level in a similar manner as `listenTCP()`

- `sendTCP(id,data,size,onSuccess)` sends `data` on a TCP connection identified by `id`. When finished, it calls the `onSuccess()` function. Note that this will hold on to the data if the TCP buffer is full, and thus delay the call to `onSuccess()`. This allows a program to queue up more data every time it re-

Figure 4.1: Abbreviated view of two-tiered implementation method used in this project



ceives the `onSuccess()` call, allowing an application like PRIME to send as much data as possible.

- `disconnectTCP(id)` simply forces a disconnect of a connection associated with that id. This will, in turn, cause the program to call the `onClose()` function associated with that connection

- `listenUDP(port,onData)` is similar to `listenTCP()`, but only has an `onData()` callback function because UDP does not involve connections

- `sendUDP(ipaddr,port,data,size,onSuccess)` is similar to `sendTCP()`, but does not require an established connection.

- `addTimer(milliseconds,onExpire,repeating)` stores either a recurring or single-use timer in the system. When this timer expires, it calls `onExpire()`. This allows an application to have periodic processes, or delayed reactions to certain events.

- `run()` is the main loop of the program. An application would call this function only once, once it has prepared the preliminary ports and timers. Using `select()`, `run()` waits for the events that it was told to monitor, and calls their respective callback functions. Note that these events are not static. While running, the callback functions can still call any of the previously mentioned functions to set up more connections, timers, etc.

- `stop()`, as its name implies, stops the server. This is the only way to cause the `run()` function to return.

This code is designed to be as flexible as possible, allowing nearly any network event-driven application to run atop it, while still meeting the needs of this PRIME

implementation. Each type of role (peer, source, and bootstrap) uses the `server` code for its low-level network interaction and timer management.

4.3 Upper Level

On top of the `server` layer, there is a different set of code for the source, peer, and bootstrap roles, stored in `source.cpp`, `peer.cpp`, and `bootstrap.cpp`, respectively. Each of these calls functions in `server.cpp`, giving different callback functions to call when events occur, such as a parent connecting, a child disconnecting, new data being sent, or a timer expiring.

The upper level also stpres the buffer, represented as a two-dimensional array, with one dimension as timestamp and the other as layer. In each cell, there is a flag to mark the state of the packet, *i.e.* whether we have the packet, we know someone who has the packet, or we haven't heard about the packet yet. Also in that space is a pointer to either the data itself, or a linked list of parents who have the data. This buffer operates as a circular array, so that every Δ seconds when the peer advances the buffer and consumes some of the data, we are not required to copy the entire structure, merely clear Δ seconds' worth of packets and advance the pointer indicating the base of the stack. A similar circular queue records which children have been informed of newly received packets. There is a pointer for each child indicating the most recent packet the child has been informed about. As children pointers advance, the former front of the queue is filled with more packets, and becomes the back of hte queue.

Most of these functions can be classified into receiver- or sender-centered tasks. The source, of course, would only contain sender functions, but the peer nodes contain both sender and receiver because of their dual role sending and receiving data, as we discussed in section 2.1.2. The bootstrap node has a unique function, and doesn't

share any code with source or peers.

Each of these functions is designed to be modular, making each function do a discrete thing that is easy to describe and reason about. This is both to make it easier to replace functions later, and to aid in understanding.

4.3.1 Scheduling Hierarchy

Consider the function call hierarchy this implementation uses when scheduling packets to be received. Each function has a discrete role, and delegates work that would complicate it.

- `assign(index,layer)` takes a packet specification, and assigns that packet to the parent with the smallest ratio of assigned bandwidth to total estimated bandwidth for this scheduling session. If there is no parent with bandwidth remaining, `assign()` will return with a failure condition.

- `schedule(minIndex,maxIndex,assignedTable,quality)` will attempt to assign all packets between `minIndex` and `maxIndex` to a parent. It attempts to assign rare packets first, and beyond that in a random order, while `assignedTable` keeps track of which packets have already been assigned. Once `schedule()` decides to attempt to schedule a packet, it calls `assign()` on that packet, leaving the decision about which parent to that function.

- `scheduleAll()` handles an entire scheduling session by calling `schedule()` on different windows, depending whether we are in the startup phase or not. First, `scheduleAll()` will call `schedule()` for the relevant windows with a `quality` that matches our current goal. Once it has gone through one round, it calls the functions again, but with the `quality` maximized, ensuring that the peer utilizes all its incoming bandwidth if possible.

4.3.2 Bandwidth Throttling

After the majority of the PRIME code was finished I discovered a problem. On the local network, there was an abundance of bandwidth, and thus the protocol cold not be properly tested because almost any system would thrive in such a rich environment. Because I did not have control over the bandwidth of the network connections, and was interfacing directly with the C sockets interface, I could not throttle my bandwidth to a reasonable, experimentally consistent level outside of the program. Therefore, I implemented a programmatic solution.

I accomplished this by having an intermediate level between the upper and lower levels. Instead of immediately sending more packets when a packet was through the `sendTCP()` function, it would instead put the IP address on a queue. Then, as often as possible, the timer would call a function called `sendToNextIPs()` to empty the queue. The peer would then wait the time it "should" have taken to send the data.

When running experiments, one can test that the program is accurately limiting the bandwidth to the specified amount.

4.4 Experimental Support

Programming PRIME itself is not the whole of this project. It also requires scripts to do various things to support experiments and analysis, including:

- Generating static overlays to test against dynamic, bootstrap-generated overlays
- Running locally several peers on each machine
- Uploading code to each PlanetLab node in our slice
- Gathering logs from PlanetLab nodes

- Analyzing the log files to determine the level of each peer
- Generating data files and graphs from individual log files

Some of these were simple scripts to automate tedious tasks, and others were much more complicated, but they all represent part of the code base of this project that hopefully will be used by another researcher.

Chapter 5

Strategies and Challenges to Performance Evaluation

An implementation of a protocol does little good if one cannot effec tively evaluate it. This chapter details the issues with evaluating a live peer-to-peer system, both in simply collecting all the data into a readable form (section 5.1), and in analyzing problems with the system (section 5.2). Section 5.3 then describes the metrics used for evaluating the success of the system, as well as for determining the source of any problems that might have arisen.

5.1 Logistical Issues

The main challenge in evaluating and debugging a peer-to-peer system is also its main advantage: its decentralized nature. Because of that, there is no one point from which we can examine the data to get an accurate picture of the system. Because we need 50 or more nodes to have a meaningful test, we need to gather logs from that many places in order to have a meaningful analysis.

A related problem is about synchronization. We need to be able to start the experiment on all nodes at one or more predefined times, and also have log files with timestamps that are consistent. Otherwise, analysis shows packets arriving at nodes beforee they have been created. To solve that problem, I used only nodes that support the Network Time Protocol (NTP), so they could all have adequately synchronized internal clocks.

5.2 Analytical Issues

The decentralized nature of the system also comes into play when analyzing the system, because the cause of a problem and its symptoms may not appear in the same place. For example, if a level 1 peer had a bug that caused it to delay for $2 * \Delta$ before requesting packets, we would not likely see the effects in that peer. We would instead see its diffusion children suffering farther down the tree. Likewise, if there were a poorly-constructed overlay, we might not see the effects anywhere in particular, just bandwidth utilization would be low because of lack of diversity. Even once we discover that there is a bug, it can be difficult to find exactly where in the code it occurs, because PRIME is a system that intentionally uses randomness to communicate with several other entities — the conditions to trigger the bug might not occur the next time that scenario is run.

5.3 Metrics for Evaluation

To evaluate the PRIME system, and to address some of the issues raised in section 5.2, we use several metrics [5] to examine what is going on in the mesh, and how well the peers are performing.

In many of these graphs, we make heavy use of the organized view discussed in section 3.1.1, and that can be seen on figure 3.1 on page 16. That is, peers are classified by their distance from the source. Sometimes this means having different graphs for peers at different levels, and sometimes this is integral to the metric itself.

When there are several data points to examine, it is usually helpful to visualize this in the form of a cumulitive distribution function (CDF). This allows us to quickly see various aspects of the data that we would not see with a cloud graph, while allowing more depth than distilling the data to a single number.

The main metrics we used for evaluating PRIME are as follows:

- **Average quality**: This is the main metric that individual clients care about. That is, what is the quality of the video that the peers receive. In the experiments, maximum quality was always 8, meaning there were 8 different descriptions for each timestamp.

- **Bandwidth utilization**: This measures how we handle the main challenge of any peer-to-peer network: effective use of bandwidth. In PRIME, this translates to the ratio of marked to data packets received by peers, since a node always uses all its bandwidth.

- **Diffusion rate**: This is the rate at which new packets enter a level of the overlay tree. This should remain fairly constant throughout the experiment, and be approximately equal to the stream rate, ensuring that all packets make it to each level. If for some reason the source node's bandwidth is below the stream rate, the diffusion rate should be equal to that.

- **Diffusion time**: This is the measure of the time it takes for new packets to reach a level of the overlay. Because new packets are prioritized, it should only

take approximmately Δ seconds for the packets to reach each subsequent level of the tree.

Chapter 6

Experiments and Results

This chapter first describes the experiments over locally-networked machines (section 6.1), detailing both the configuration and results of the tests. Then, section 6.2 describes the configuration of PlanetLab and the tests thereon. Finally, I reveal the results of my tests on PlanetLab. Higher-quality versions of these graphs can be found in Appendix A.

6.1 Local experiments

6.1.1 Configuration

For the local tests, twelve 2.16 GHz Intel Core 2 Duo iMacs with 3 GB of RAM running Mac OS 10.5.7 each ran 5 peers. One 2.20 GHz Intel quad-core Linux machine with 500 GB of RAM, running Fedora Core 9 ran the bootstrap and source nodes. In total, there were 60 peers, one bootstrap, and one source running on a local network

For these tests, each packet was 640 bytes, representing 50 ms of one layer of the stream. There were 8 possible layers, which adds up to a stream rate of 800 kbps (or 100 kBps). The maximum number of children a node could have was 6, and that

Table 6.1: Configurations used in local experiments

Pkt size (*bytes*)	640	640	640	5120	640
Pkt time (*ms*)	50	50	50	50	50
Max quality	8	8	8	8	8
Rate (*kbps*)	800	800	800	800	800
Peer bw (*kbps*)	835.92	835.92	835.92	6437.52	835.92
Source bw (*kbps*)	835.92	875.04	875.04	7000.32	875.04
Δ (*sec*)	5	5	5	5	5
ω (Δs)	7	7	7	8	7
Node degree	6	6	6	6	6
Duration (*sec*)	600	600	600	600	600
Overlay	rand	file	file	file	rand

was the number of parents a node was trying to acquire. This leaves an optimum bandwidth of 133 $^1/_3$ kbps (or 16 $^2/_3$ kBps) per connection. Source bandwidth was varied, but peers' outgoing bandwidth was kept equal to the stream rate, plus a small amount for overhead associated with piggybacking packet announcements (see page 17). ω was also varied between 7Δ and 8Δ. The experiments lasted 10 minutes each, or 120 Δ intervals.

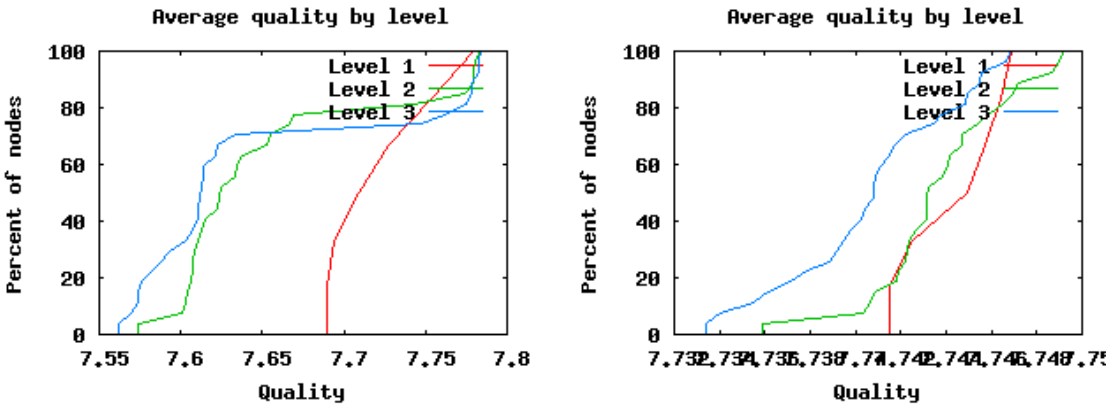
6.1.2 Results

The results from running local experiments generally matched the ns simulation results [5] with one notable exception in efficiency for a few seconds after startup phase ends. That is discussed at the end of this subsection.

Change in ω

The maximum depth for local tests was 4, which would set $\omega = 7 * \Delta$. When varying to $\omega = 8 * \Delta$, there was an increase in average quality, but only slightly, as shown in figure 6.1.

Figure 6.1: Comparing quality by ω



(a) $\omega = 7 * \Delta$ (page 48)

(b) $\omega = 8 * \Delta$ (page 48)

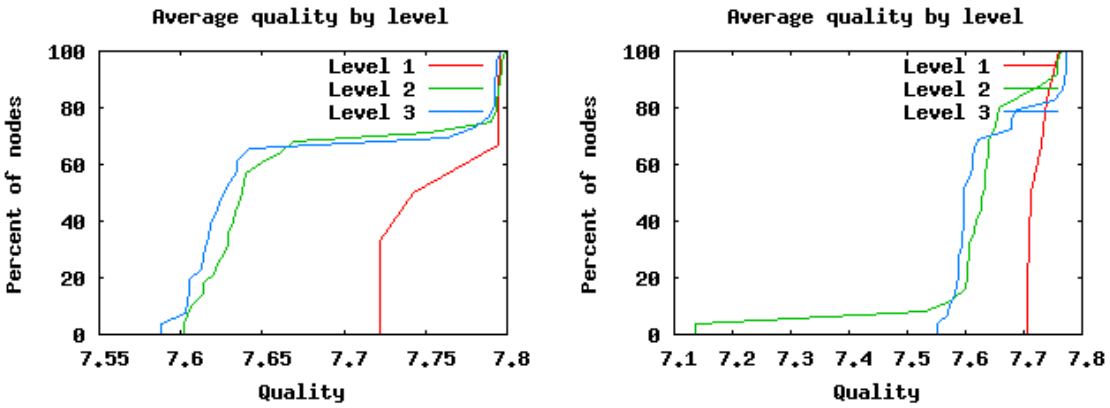
Predefined vs. Bootstrap-defined Overlay

Some tests ran with a predefined overlay designed to distribute content effectively, which means that each peer was told its parents beforehand. Comparing this to random overlays generated by the bootstrap node on the fly, there is no significant difference in average quality. The only noticeable trait is that in the random case, peers took longer to connect to steady parents, but this was before startup phase began, so it had no effect on transmission.

Source Bandwidth changes Received Quality

When changing the source bandwidth, we expect to see a higher quality sent by the source. When that translates into the source having fewer packets at the beginning that are never sent, we expect to see a commensurate increase in peer received quality. Conversely, when this translates into more duplicate packets being sent out, we only expect to see a marginal increase in peer received quality, because hopefully those duplicate packets were already being swarmed around. Comparing the quality sent and received CDFs (per timestamp, not average as in figure 6.3) shows us that

Figure 6.2: Comparing quality by overlay type



(a) Predefined Overlay (page 49)

(b) Bootstrap (page 49)

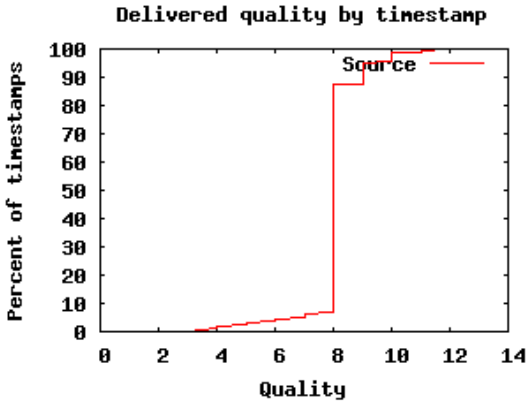
switching from 0% extra to 5% extra is indeed a shift in source output, but the shift in peer input occurs mainly when the quality was already 5 or higher.

Dip in efficiency

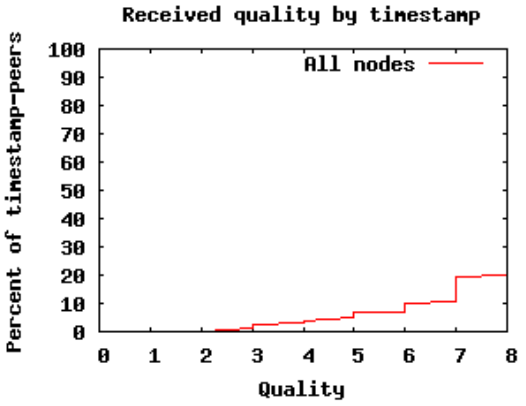
There was one property that all runs had that was unexpected, deviating from the simulation results. Immediately after the startup phase, there is a severe drop in bandwidth efficiency in all connections except connections with source. This is only visible if you examine efficiency over time, as in figure 6.4. When examining the logs, we see that this is due to the peers not being able to schedule the whole 5 seconds (Δ) of data directly after the normal phase begins. At this time, I am unable to determine the cause of this anomaly, and the reason it did not appear in simulation results. It represents a very small drop in overall quality, so this does not seriously threaten PRIME's effectiveness, but nevertheless it should be identified and eliminated in the code's next iteration.

Figure 6.3: Comparing quality by source bandwidth

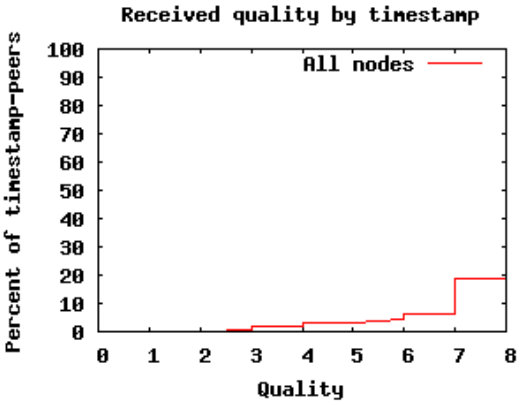
(a) Source output with 0% extra bandwidth (page 50)



(b) Source output with 5% extra bandwidth (page 50)

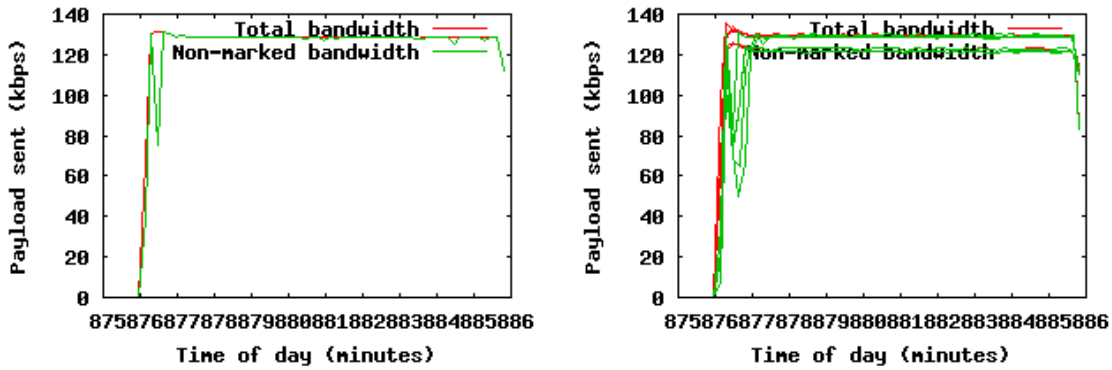


(c) Total input with 0% extra bandwidth (page 51)



(d) Total input with 5% extra bandwidth (page 51)

Figure 6.4: Dip in efficiency for most connections



(a) Diffusion connections for a level 2 peer (page 52)

(b) Swarming connections for a level 1 peer (page 53)

6.2 PlanetLab

6.2.1 Configuration

PlanetLab [6] is a set of computers spanning the globe intended for developing and testing planetary-scale services. To use it, a researcher get a "slice" from his school's site coordinator, and then adds nodes that slice, giving the researcher "slivers" on each node on which to run experiments. For this study, I first acquired several hundred nodes, the ran small tests to determine if they were suitable for these experiments. After removing those that failed to respond to login attempts or install basic software, I was left with approximately 150 nodes, all of which ran a single peer for each experiment. The source and bootstrap nodes were hosted locally on the aforementioned Linux computer.

For the PlanetLab tests, all factors affecting the stream rate were identical to the local tests. The degree was varied between 4, 6 and 8. Source bandwidth remained capped at the stream rate, but often the realities of the Internet gave the source node a lower bandwidth than the stream rate, resulting in lower quality for the entire

Table 6.2: Configurations used in PlanetLab experiments

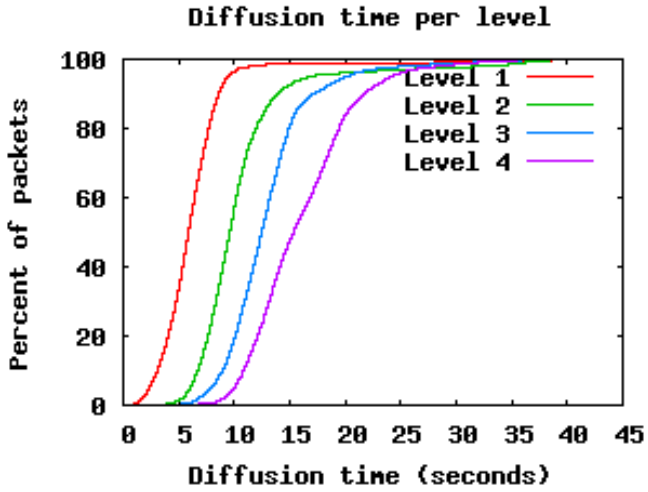
Pkt size (*bytes*)	640	640	640
Pkt time (*ms*)	50	50	50
Max quality	8	8	8
Rate (*kbps*)	800	800	800
Peer bw (*kbps*)	835.92	835.92	835.92
Source bw (*kbps*)	835.92	835.92	835.92
Δ (*sec*)	5	5	5
ω (Δs)	8	8	8
Node degree	4	6	8
Duration (*sec*)	600	600	600
Overlay	rand	rand	rand

swarm. ω remained at 8, although in normal circumstances, ω would vary with degree, because it changes the likely depth of the mesh. Like the local tests, these experiments lasted 10 minutes each, or 120 Δ intervals.

6.2.2 Results

Results from the PlanetLab runs were much more heterogeneous than those from the local test. Peers often had lower bandwidth than was being throttled, which allowed other connections to have very high bandwidth, as only the aggregate bandwidth is throttled, not per connection. To manage heterogeneous bandwidth, the PRIME protocol attempts to enforce a common per-connection speed. High-bandwidth users, instead of having potentially higher-bandwidth (and potentially not) connections, simply connect with more peers. A low-bandwidth peer might have only enough bandwidth to connect to one or two peers. Unfortunately, this implementation assumed the ability to control bandwidth, so this functionality was not implemented.

Figure 6.5: Diffusion times over PlanetLab (page 55)



Base Performance

Despite the heterogeneity of results, several of the metrics measured results perfectly in line with expectations. Diffusion time is one such measurement. Most packets reach the next level in 5 seconds or less, as shown in 6.5

The logs show that the source sent only 82% of the packets, a fact that agrees with the diffusion rate graph on page 56, which shows the diffusion rate for level 1 peers as between 650 and 700 kbps. Given that, we expect average quality to be 6.5 or worse. Figure 6.6 shows that efficiency is not that high.

As is evident in figure 6.7, the main inefficiencies lie in high-bandwidth connections not being effectively used. On the low end of bandwidth, total and useful bandwidth are nearly the same, but once total bandwidth increases, effective bandwidth usually cannot raise to the same level. Implementing a system where peer degrees are dynamic would likely fix this inefficiency.

Figure 6.6: Average received quality over PlanetLab (page 54)

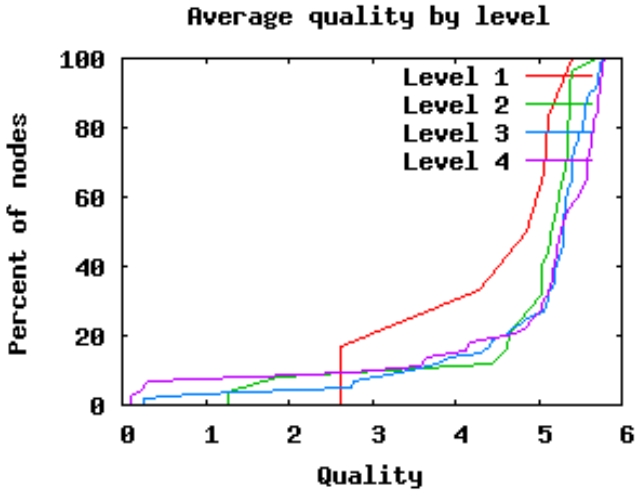
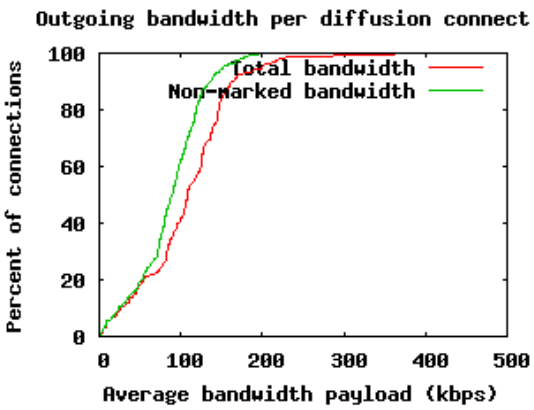
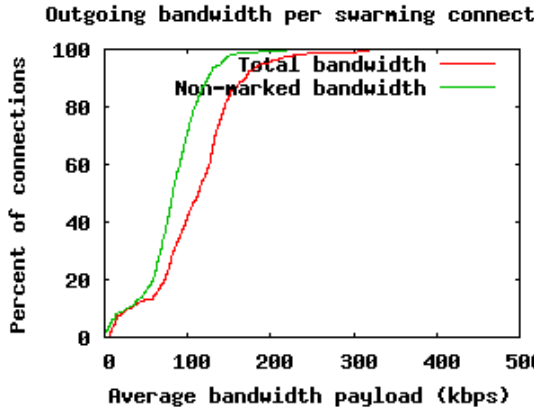


Figure 6.7: Bandwidth utilization over PlanetLab

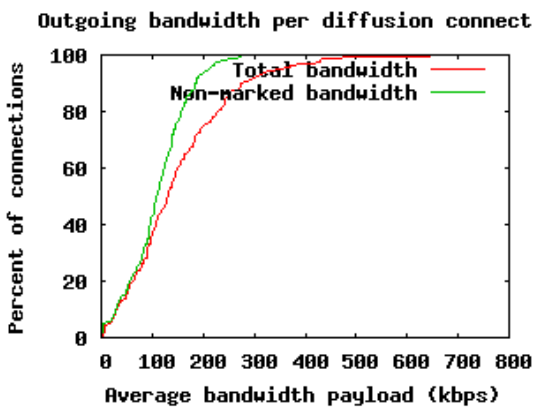


(a) For diffusion connections (page 57)

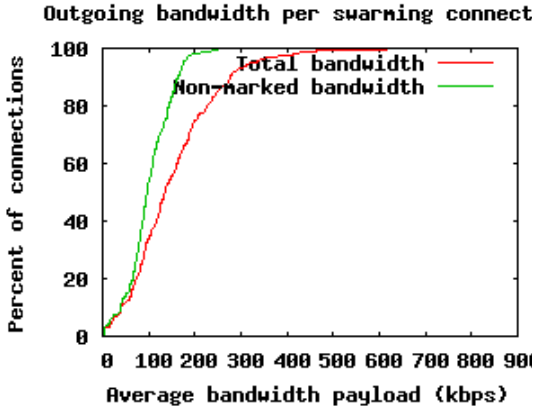


(b) For swarming connections (page 58)

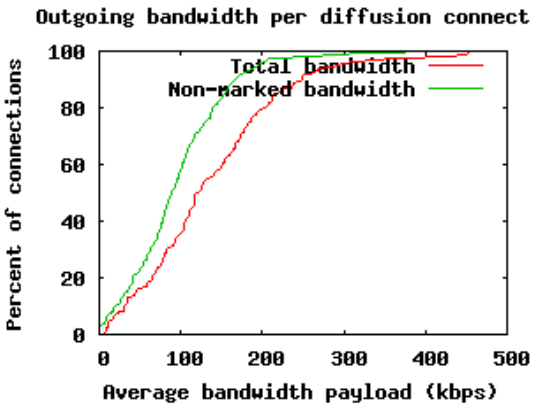
Figure 6.8: Bandwidth utilization over PlanetLab



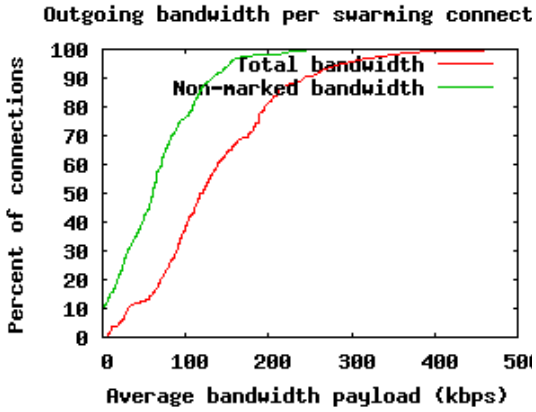
(a) For degree 4 diffusion connections (page 56)



(b) For degree 4 swarming connections (page 57)



(c) For degree 8 diffusion connections (page 58)



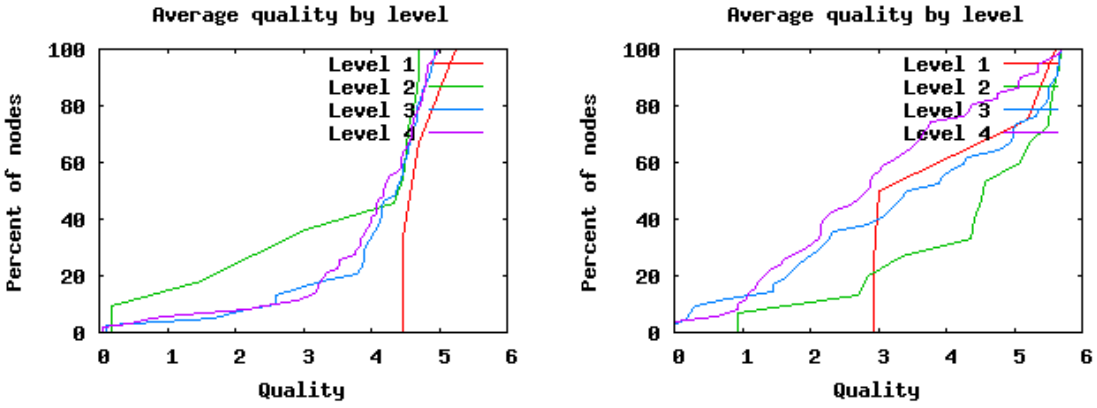
(d) For degree 8 swarming connections (page 59)

Effect of Peer Degree

Comparing the efficiency of the system using degree 6 in figure 6.7 with degrees 4 and 8 in figure 6.8, degree 8 is clearly the least efficient, and degree 4 appears to be most efficient.

When examining the average received quality for each trial, one can see a clearer picture. A 4-child system is much better for those at degree 1 because they each can use more of the source's bandwidth. Those that are in level 2, for example, benefit from the degree 8 trial. Degree 6 seems to have the best overall quality.

Figure 6.9: Average received quality over PlanetLab



(a) Degree 4 (page 54)

(b) Degree 8 (page 55)

Unfortunately, there were not enough trials to reach any conclusiions as far as the best static peer degree, but they did generate some hypotheses. These trials also showed that PRIME can work in the field, and that it needs higher bandwidth for the source and heterogeneous peer degree.

Chapter 7

Conclusion

This paper shows the challenges of live, peer-to-peer streaming, and how PRIME handles them. It introduces one PRIME impementation, and the challenges inherent therein. It shows the ways in which one would evaluate the PRIME protocol, and evaluates several experimental trials. With one notable exception, these experiments matched expectations and the simulation tests. I believe this system is fit for further development, building on top of the current code base. We are now one step closer to realizing the goal of peer-to-peer live streaming, and getting it into the hands of consumers. Here are the next steps toward that goal, in no particular order:

- Pair this with a video encoding method, so that we could be actually streaming video, instead of dummy packets. (The code is marked in every place it would need to be changed for that)
- Add the method for having a dynamic node degree based on bandwidth. This would accommodate the varying bandwidths of consumers on the Internet.
- Test further on PlanetLab, to ensure that the system has the properties we want over such a heterogenous space.

- Finally, build a user-friendly interface so that anyone can use it.

If we can do that, we could be well on our way to a more open Internet.

Appendix A

Full-Quality Graphs

Figure A.1: Average quality with $\omega = 7 * \Delta$, source having 5% extra bandwidth on a predefined overlay

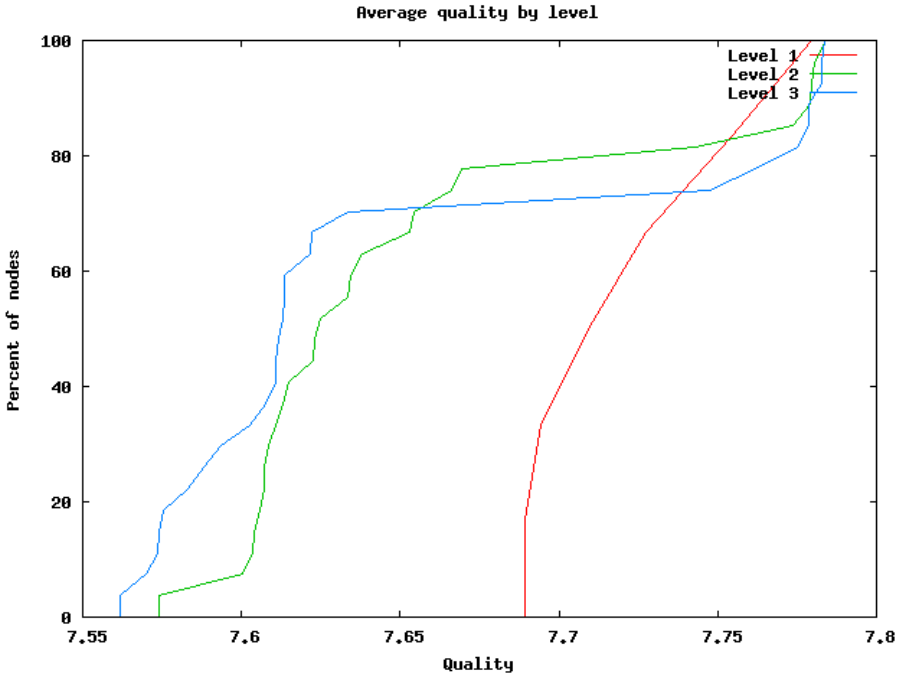


Figure A.2: Average quality with $\omega = 8 * \Delta$, source having 5% extra bandwidth on a predefined overlay

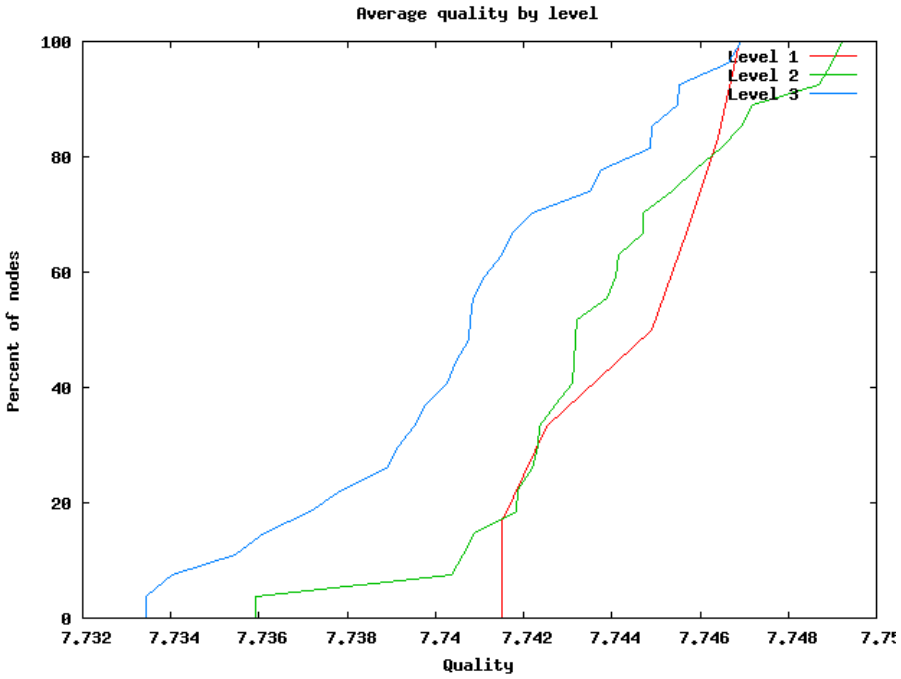


Figure A.3: Average quality with predefined overlay, source having 5% extra bandwidth and $\omega = 7 * \Delta$ on a random mesh

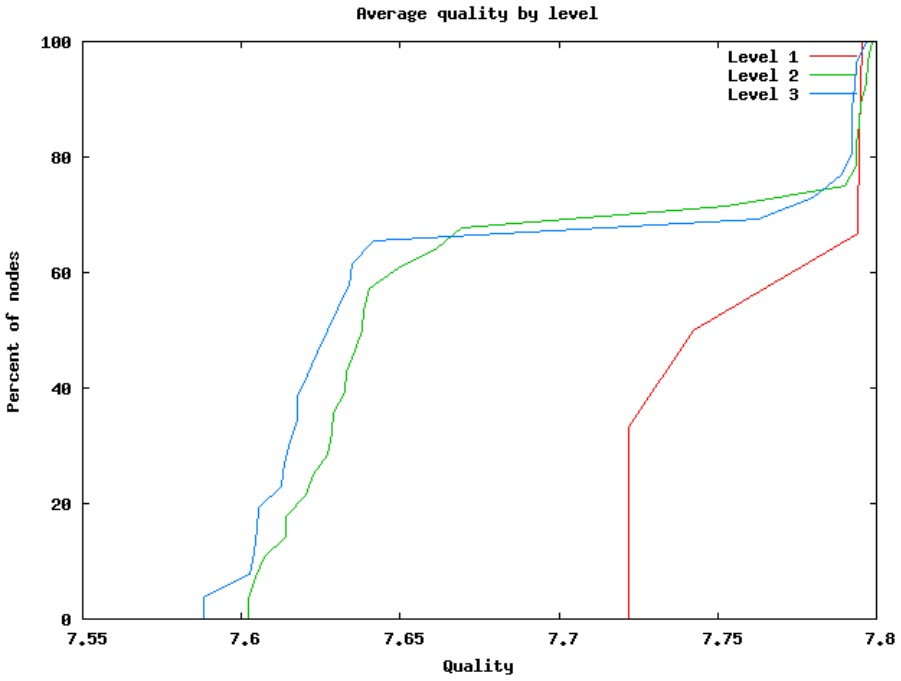


Figure A.4: Average quality with a random mesh, source having 5% extra bandwidth and $\omega = 7 * \Delta$ on a random mesh

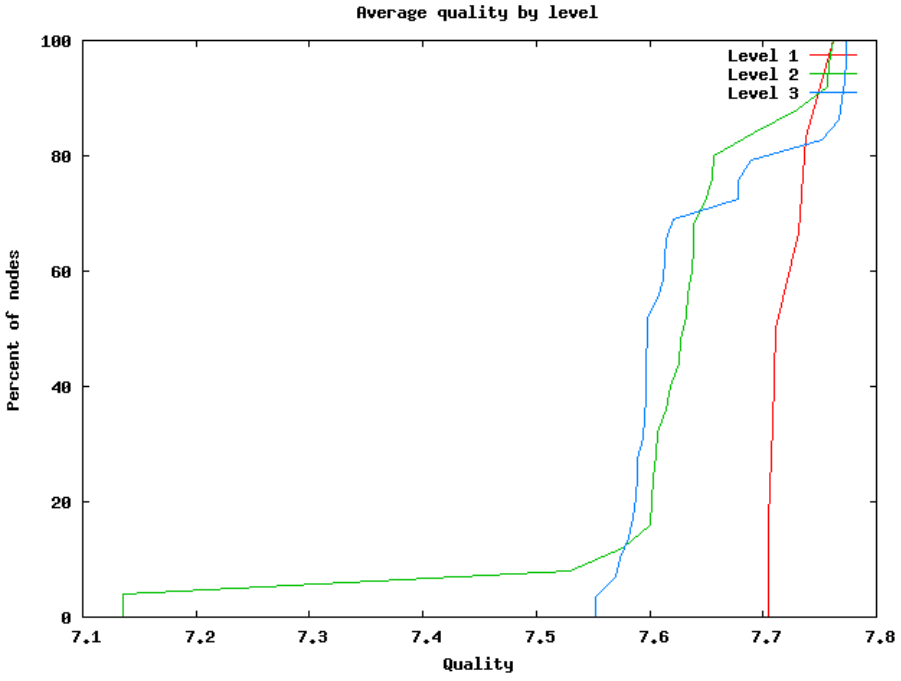


Figure A.5: Source sent quality per timestamp with source having 0% extra bandwidth and $\omega = 7 * \Delta$ on a random mesh

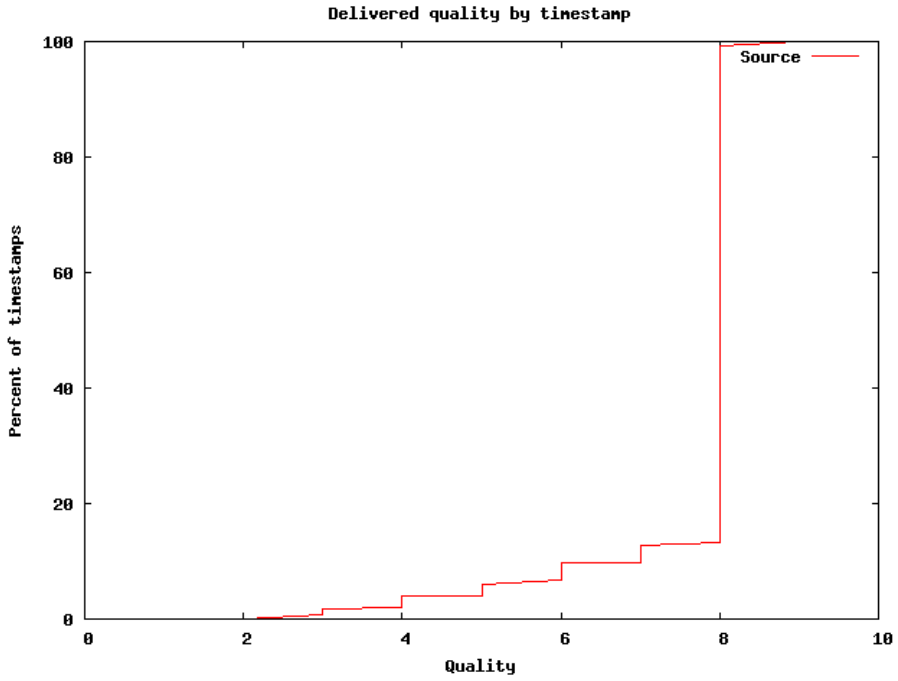


Figure A.6: Source sent quality per timestamp with source having 5% extra bandwidth and $\omega = 7 * \Delta$ on a random mesh

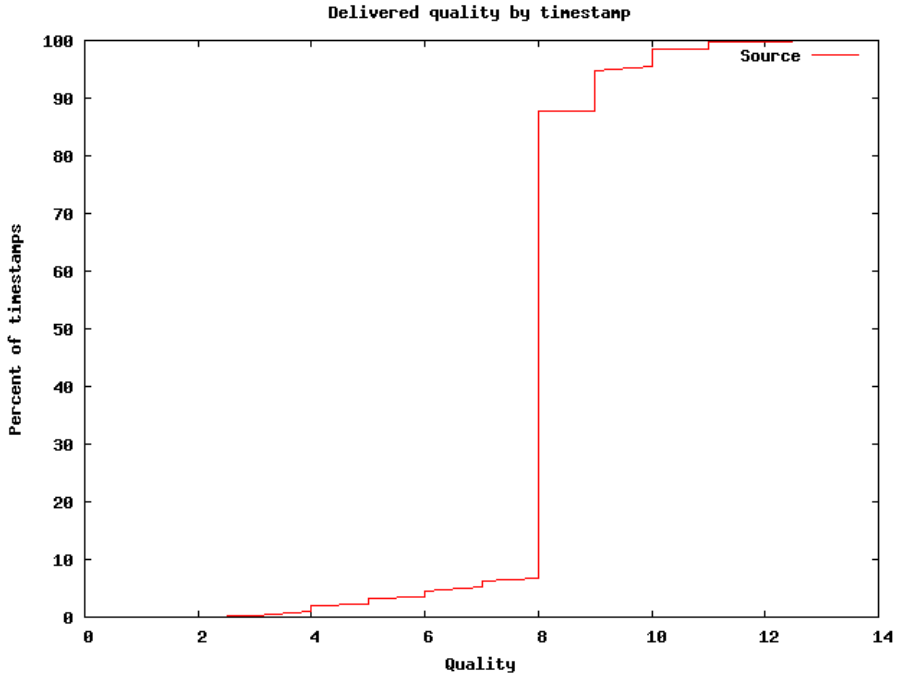


Figure A.7: Quality per timestamp with source having 0% extra bandwidth and $\omega = 7 * \Delta$ on a random mesh

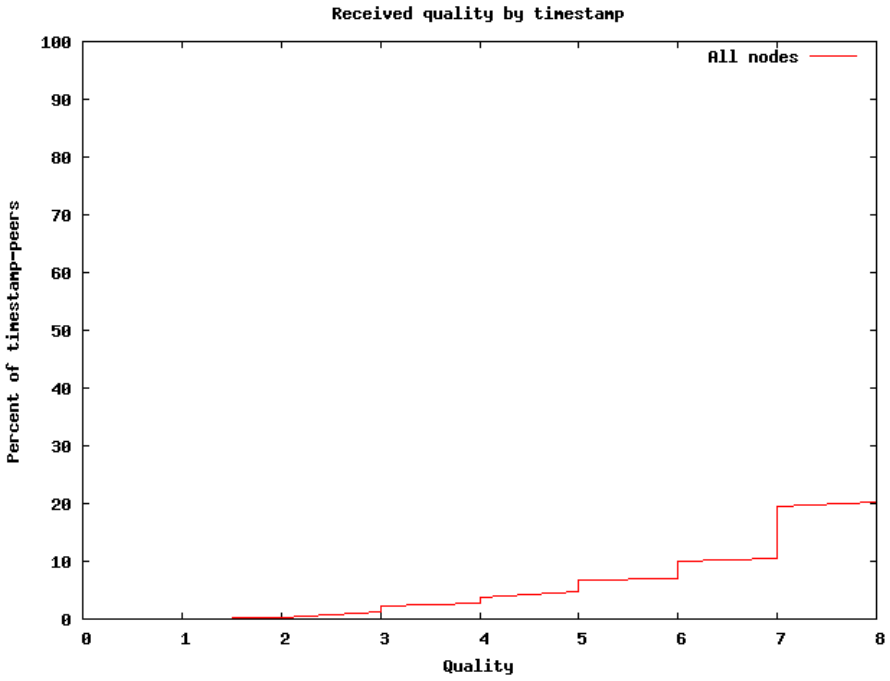


Figure A.8: Quality per timestamp with source having 5% extra bandwidth and $\omega = 7 * \Delta$ on a random mesh

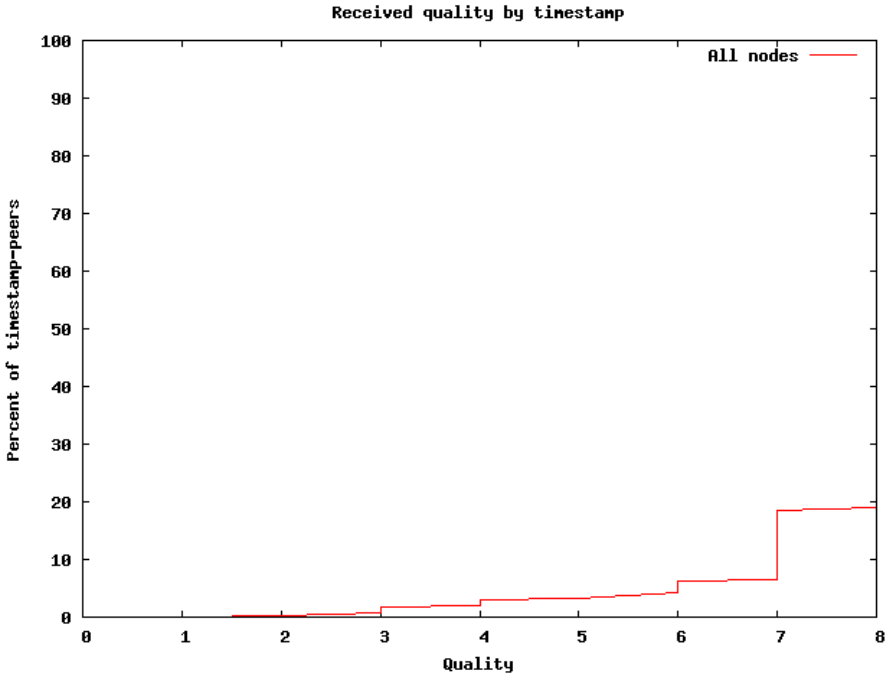


Figure A.9: Bandwidth utilization for the diffusion connections of a level 1 peer, source having 0% extra bandwidth and $\omega = 7 * \Delta$ on a random mesh

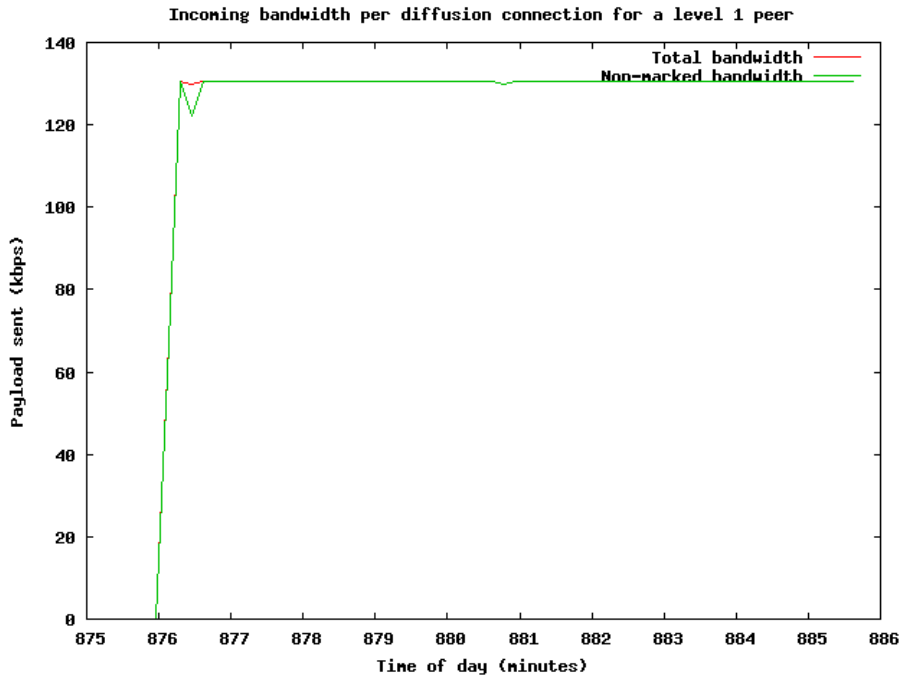


Figure A.10: Bandwidth utilization for the diffusion connections of a level 2 peer, source having 0% extra bandwidth and $\omega = 7 * \Delta$ on a random mesh

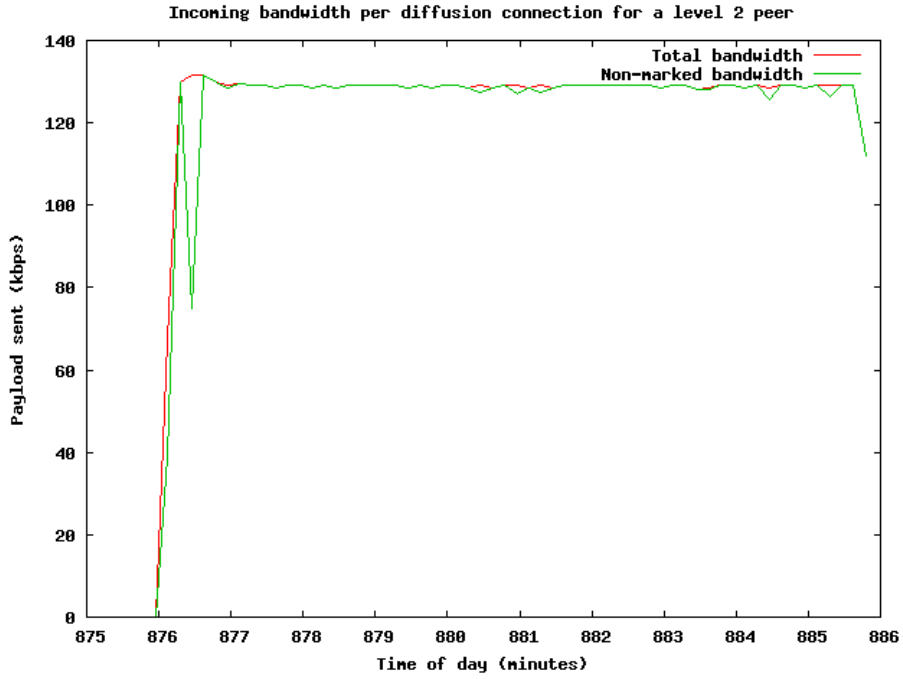


Figure A.11: Bandwidth utilization for the swarming connections of a level 1 peer, source having 0% extra bandwidth and $\omega = 7 * \Delta$ on a random mesh

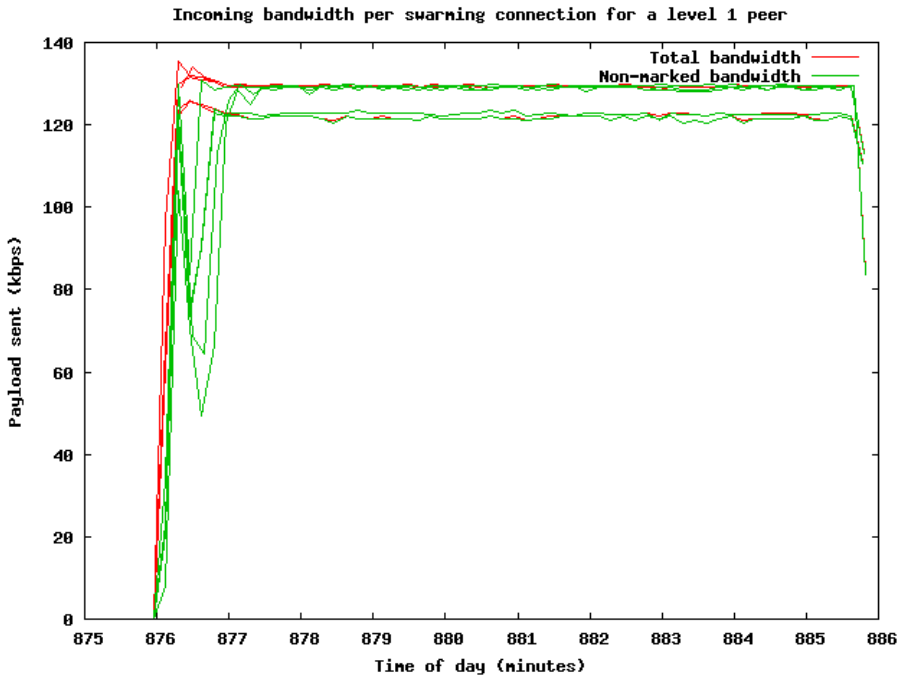


Figure A.12: Bandwidth utilization for the swarming connections of a level 2 peer, source having 0% extra bandwidth and $\omega = 7 * \Delta$ on a random mesh

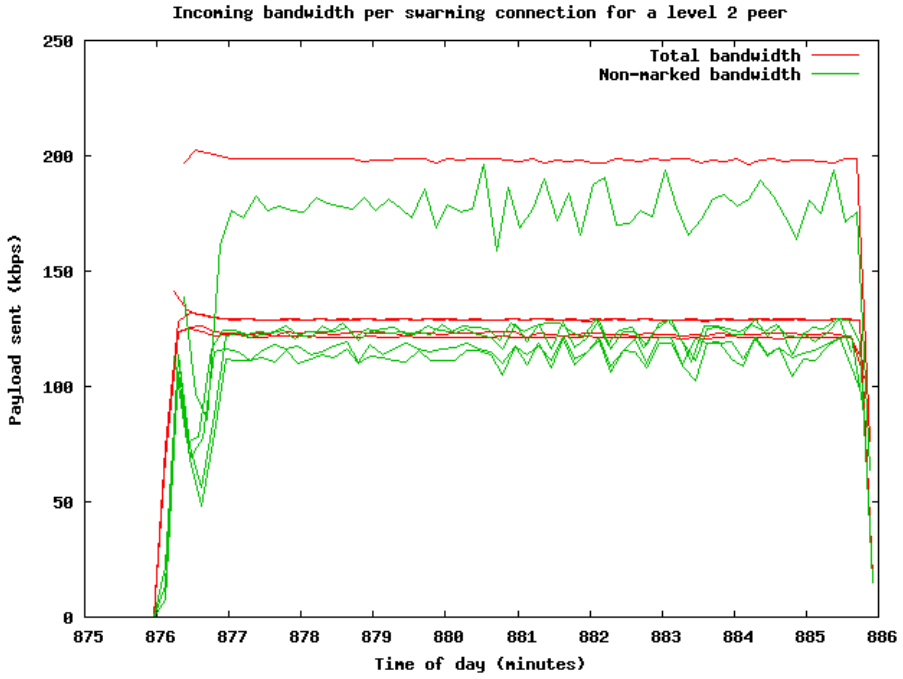


Figure A.13: Average received quality on PlanetLab with degree 4

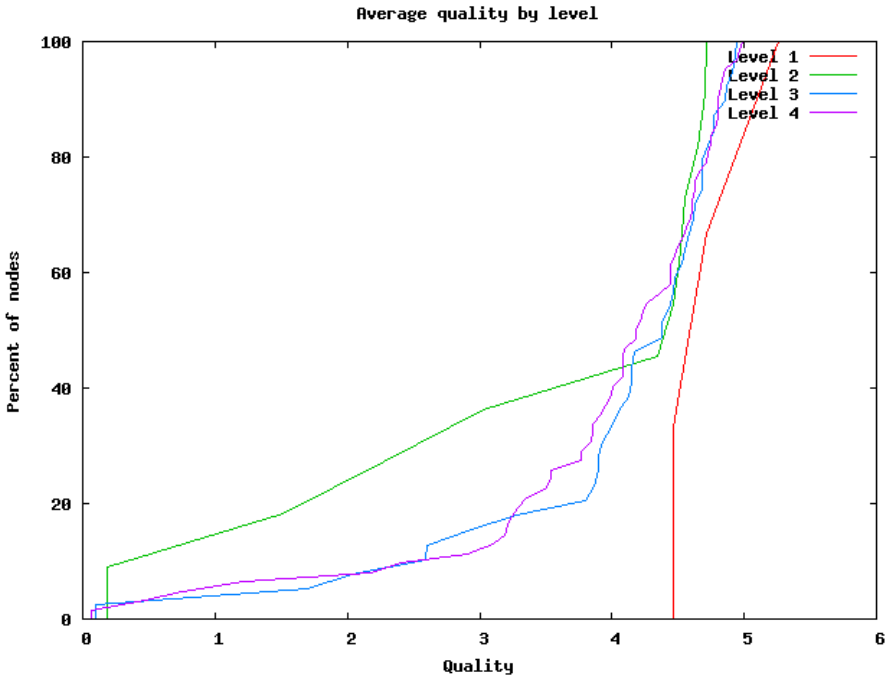


Figure A.14: Average received quality on PlanetLab with degree 6

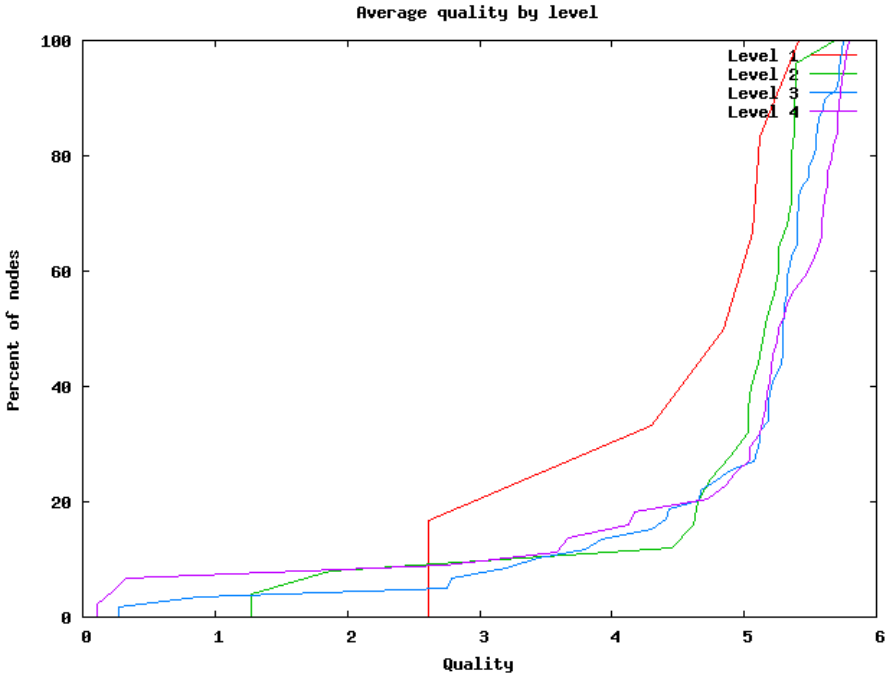


Figure A.15: Average received quality on PlanetLab with degree 8

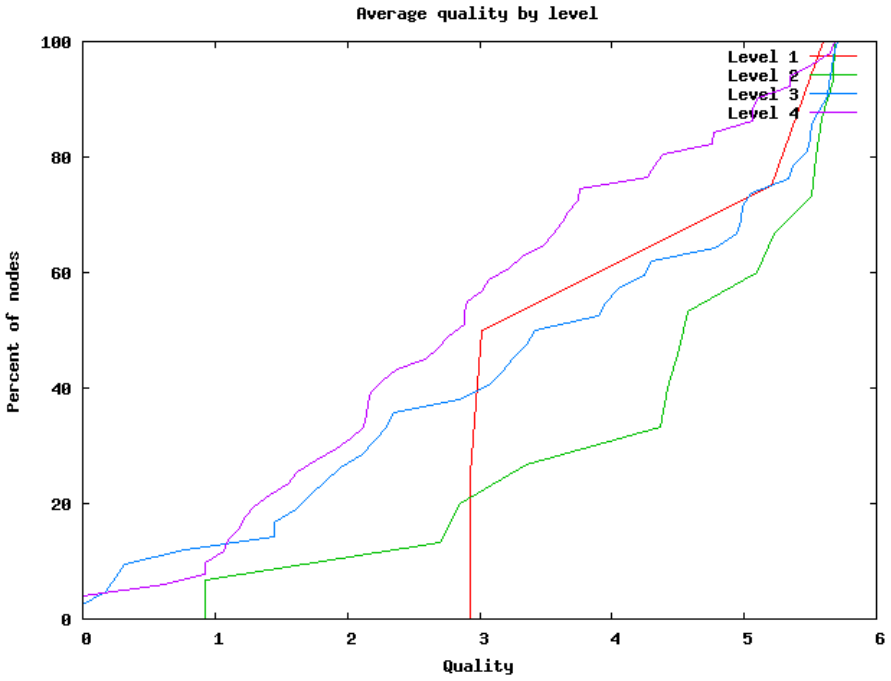


Figure A.16: Diffusion times on PlanetLab with degree 6

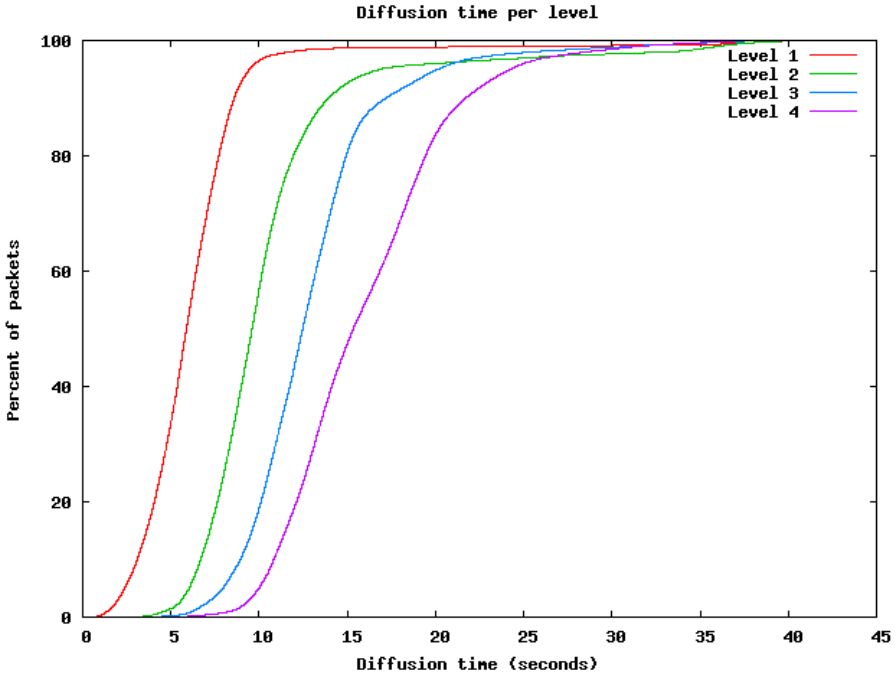


Figure A.17: Diffusion rates on PlanetLab with degree 6

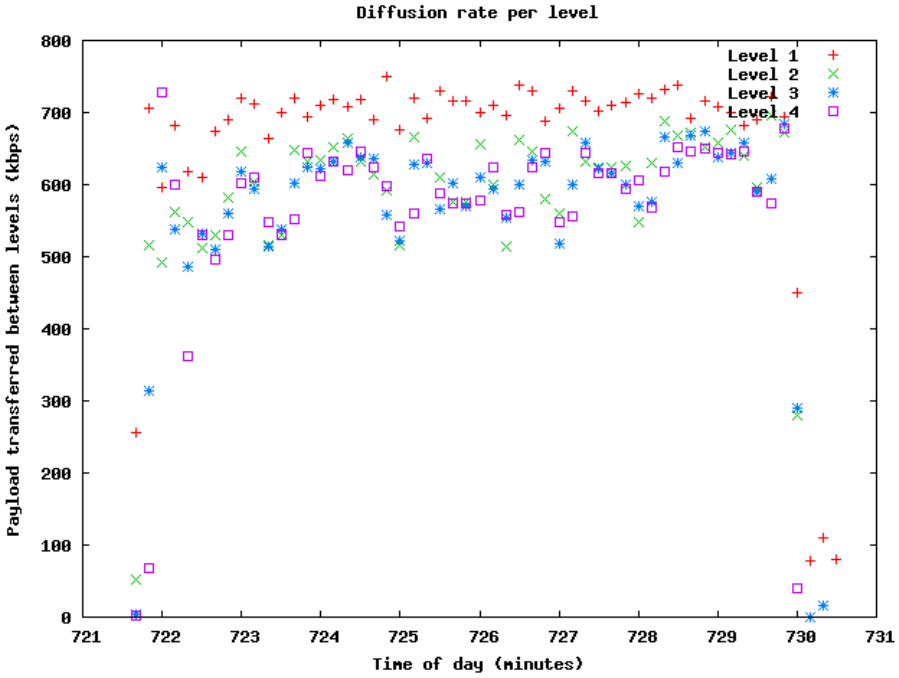


Figure A.18: Bandwidth utilization for diffusion connections on PlanetLab with degree 4

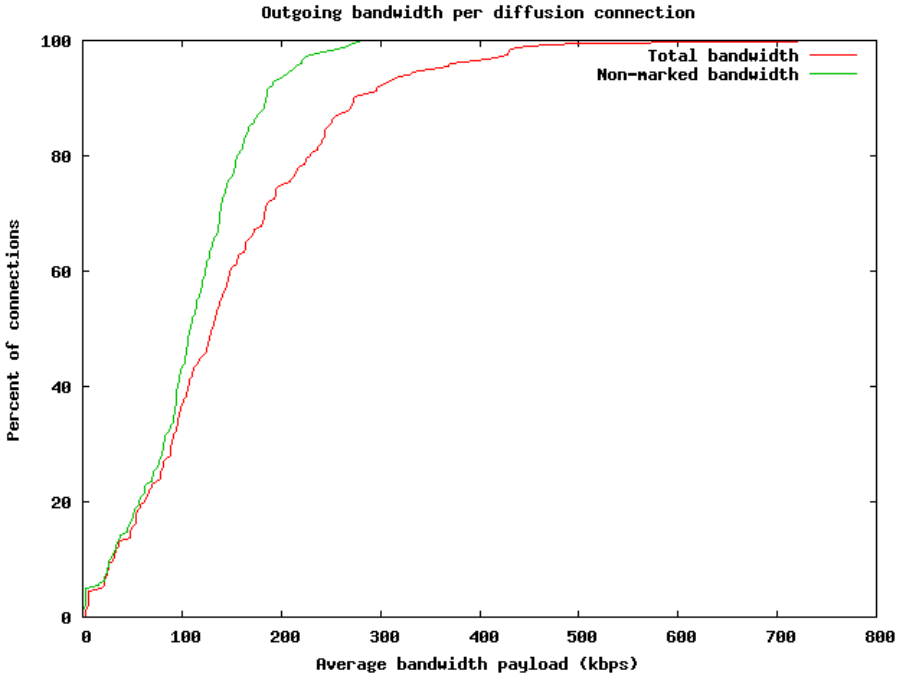


Figure A.19: Bandwidth utilization for swarming connections on PlanetLab with degree 4

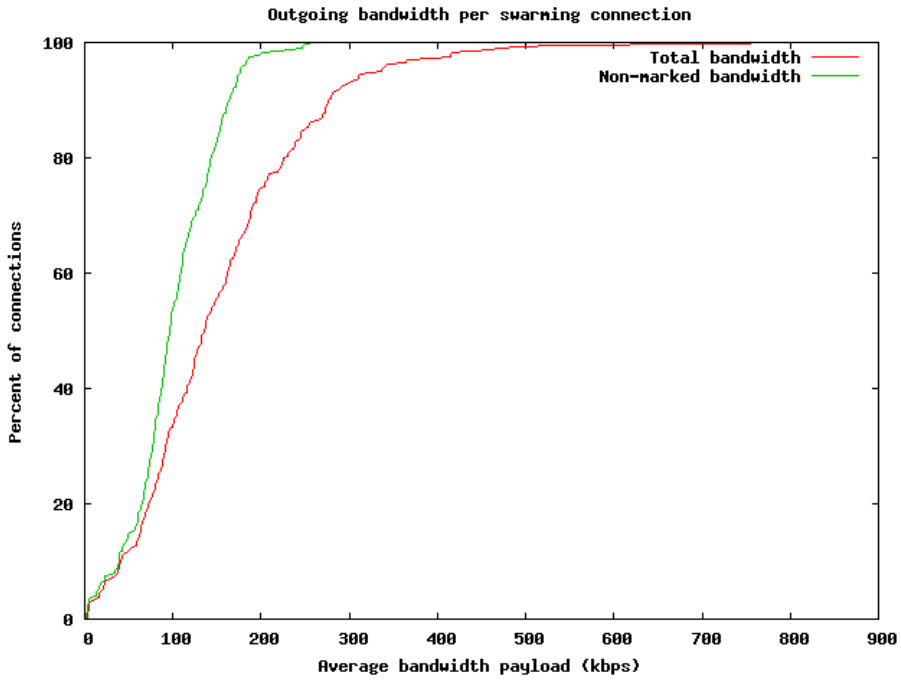


Figure A.20: Bandwidth utilization for diffusion connections on PlanetLab with degree 6

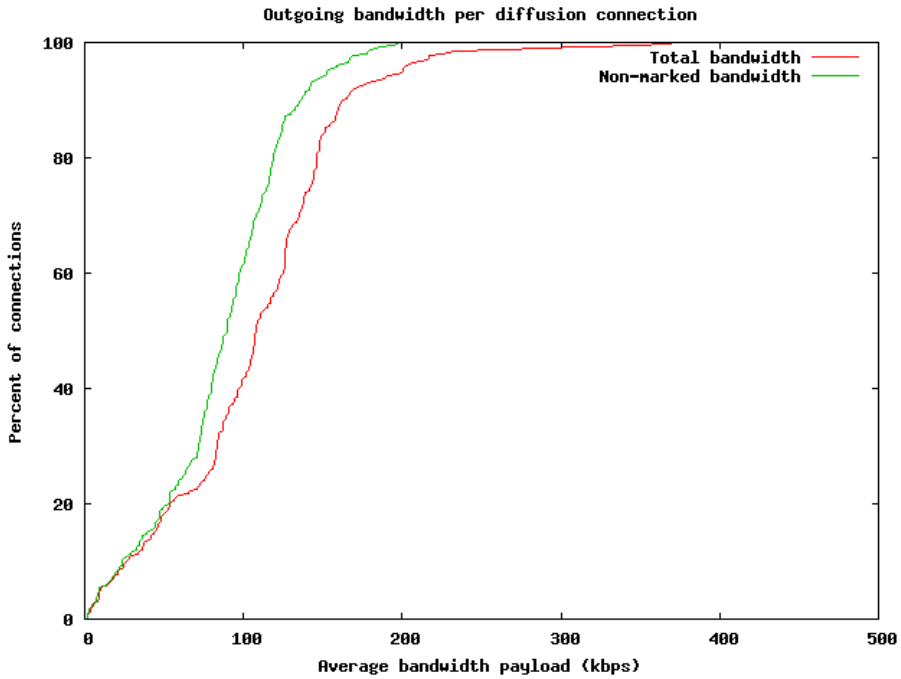


Figure A.21: Bandwidth utilization for swarming connections on PlanetLab with degree 6

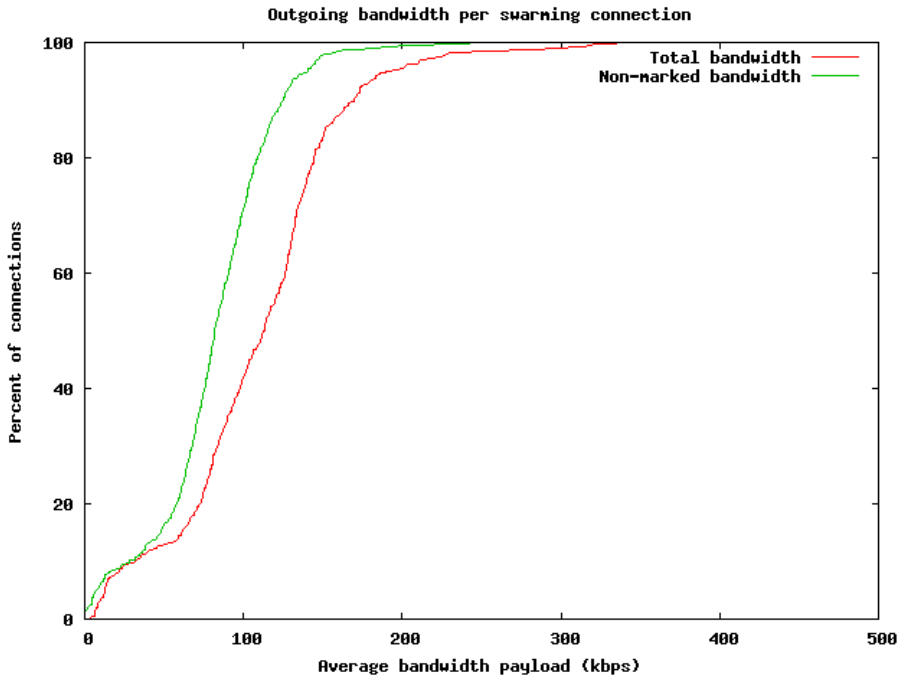


Figure A.22: Bandwidth utilization for diffusion connections on PlanetLab with degree 8

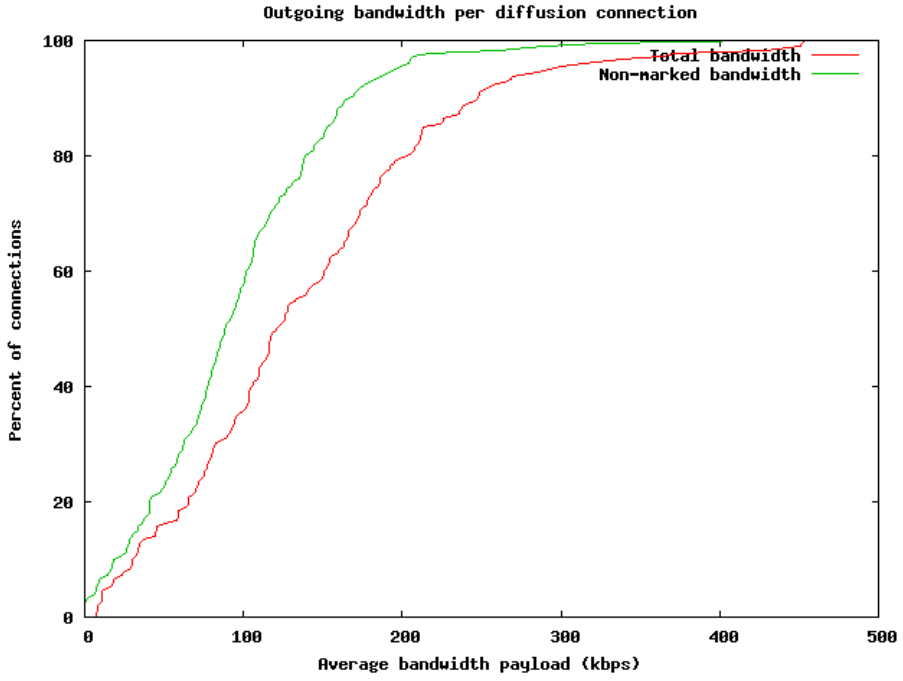
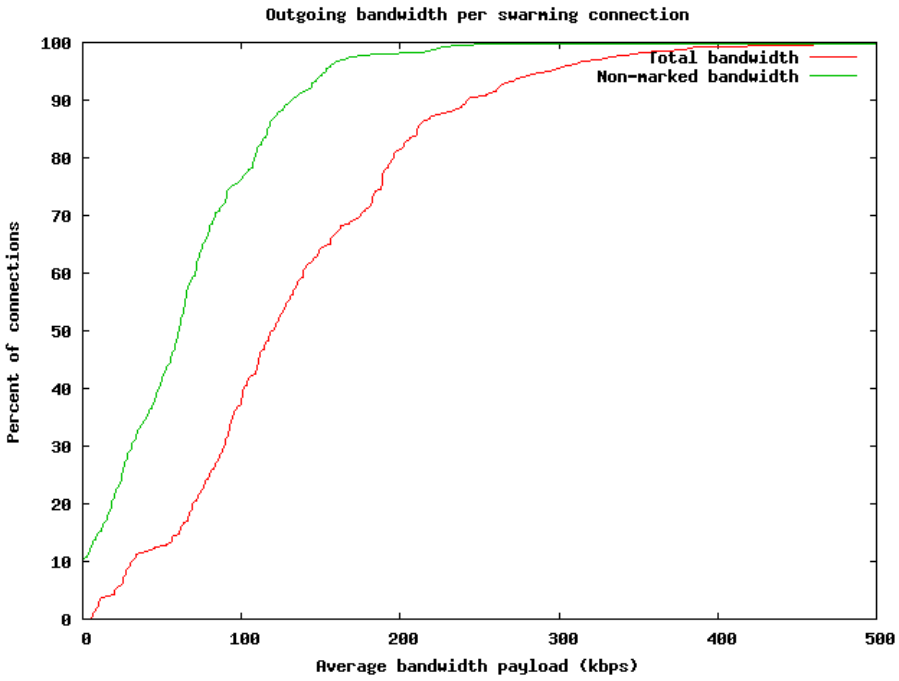


Figure A.23: Bandwidth utilization for swarming connections on PlanetLab with degree 8



Bibliography

[1] B. Cohen, "Bittorrent." [Online]. Available: http://www.bittorrent.com

[2] B. Chang, "Rapidshare." [Online]. Available: http://www.rapidshare.com

[3] V. N. Padmanabhan, H. J. Wang, and P. A. Chou, "Resilient peer-to-peer streaming," in *ICNP*, 2003.

[4] N. Magharei and R. Rejaie, "PRIME: Peer-to-Peer Receiver-drIven MEsh-based Streaming," in *INFOCOM*, 2007.

[5] ——, "Dissecting the performance of Live Mesh-based P2P Streaming," University of Oregon, Tech. Rep. CIS-TR-07-05, 2007. [Online]. Available: http://mirage.cs.uoregon.edu/pub/tr07-05.pdf

[6] "Planetlab." [Online]. Available: http://www.planet-lab.org

[7] X. Zhang, J. Liu, B. Li, and T.-S. P. Yum, "Coolstreaming: A data-driven overlay network for live media streaming," in *INFOCOM*, 2005.

[8] V. Venkataraman, K. Yoshida, and P. Francis, "Chunkyspread: Heterogeneous Unstructured End System Multicast," in *ICNP*, 2006.

[9] N. Magharei, R. Rejaie, and Y. Guo, "Mesh or Multiple-Tree: A Comparative Study of P2P Live Streaming Services," in *INFOCOM*, 2007.

[10] "Network simulator – ns (version 2)," *Software*, 2002. [Online]. Available: http://www.isi.edu/nsnam/ns/