Design and Evaluation of Load Balancing Algorithms in P2P Streaming

WANG, Yongzhi

A Thesis Submitted in Partial Fulfilment
of the Requirements for the Degree of
Master of Philosophy
in
Information Engineering

The Chinese University of Hong Kong
August 2009

Abstract of thesis entitled:
Design and Evaluation of Load Balancing Algorithms in P2P Streaming
Submitted by WANG, Yongzhi
for the degree of Master of Philosophy
at The Chinese University of Hong Kong in August 2009

In this thesis, we consider a central problem in P2P content distribution: given a set of neighboring peers connected to each other to exchange content, how they can optimally negotiate the rate in helping each other so as to achieve maximum overall throughput and minimize the content server's load. We call this the "load balancing problem" of a P2P system. By providing an abstract formulation of the optimization problem, we contrast this problem with the network congestion control problem, both in terms of parallels and differences. We then proceed to study several versions and aspects of this problem systematically: (a) request allocation, (b) neighbor selection, and (c) server load minimization. We have proposed and evaluated several practical algorithms that are discrete (window-based), distributed (without needing global information), and adaptive.

中文摘要

在这篇论文中，我们研究了分布式对等网络中的一个核心问题：给定一些可用的邻居节点后，各个节点之间如何协商互相之间的下载速率，使得节点的上行带宽的利用率达到最高，并且使得服务器提供的带宽最小。我们称这个问题为，分布式对等网络中的负载均衡问题。首先，利用最优化理论，我们建立了该问题的数学模型。然后，我们分析了该问题和网络中的拥塞控制问题的相同点和不同点。最后，我们系统性的研究了负载均衡问题的三个子问题：(1)请求分配问题，(2)邻居节点选择问题，(3)服务器的负载最小化问题。我们设计并且评估了一系列实用的分布式自适应性算法，来解决负载均衡问题。这些算法都可以直接应用于实际的分布式对等网络中，而且不会在网络中产生额外的信令开销。

Acknowledgement

First of all, I would like to thank my supervisor, Prof. Chiu Dah Ming, for his encouragement of free thinking and valuable guidance during my whole research period. I have learnt so much from the supervision of Prof. Chiu and I would like to thank him for his support and trust in the past two years.

Secondly, I would like to thank Prof. John, C. S. Lui. From his courses, I have learnt the fundamental knowledge of modeling and analysis in the computer networking area. Regular meetings with Prof. Lui also provides valuable advice on my research.

Thirdly, I would like to thank all my friends in CUHK, especially Fu Zhengjia, who gave me help and support. With you around, my life in CUHK is wonderful.

As always, I would like to thank my girl friend and family for their endless love, understanding, support and encouragement through the whole process.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A Peer-to-Peer (P2P) content distribution system, whether streaming or file downloading, relies on using multiple distribution trees (not necessarily built explicitly) from the source to all peers simultaneously, to maximize throughput. These distribution trees are logical (overlay) trees tracing out which peers are helping the source to forward content to which other peers, and they share the same physical network. It is therefore necessary to adjust the data rates in these distribution trees to maximize throughput, by increasing the rate in those trees that have free (physical) capacity, and controlling the rate in those trees that are saturated. How to achieve this by distributed algorithms executed by peers is the topic of this thesis.

The difference between streaming and downloading is that the former needs to achieve a given playback rate whereas the latter means best-effort. In steaming, the content has deadlines to meet, so the delivery system needs to take them into consideration. Other than that, streaming is an easier problem than downloading when the total bandwidth capacity of the multiple distribution trees is abundant (or at least adequate) compared to the playback rate. In this thesis, we focus on the P2P streaming problem.

Broadly speaking, there are two types of P2P systems: structured versus unstructured, referring to two very different ap-

proaches of building and maintaining the multiple distribution trees from the source to all the peers. In the structured approach, the trees are built explicitly - each peer knows its role (or position) in each tree that it belongs to. Since peers come and go, maintaining these trees becomes complicated. In the unstructured approach, trees are created on-demand, based on availability of content for exchange by neighboring peers. Each peer keeps track of a subset of other peers as neighbors. At any moment, each peer tries to stream a *chunk* of content from a subset of neighbors that have this chunk. The local peer can request different pieces of the chunk from different neighbors at the same time. Thus, each piece travels through potentially a different tree to reach all peers. While building distribution trees in this manner seems rather chaotic, it works incredulously well, as demonstrated by many practical and deployed systems (such as BitTorrent [1] and PPLive [26]).

The throughput limit of such multiple-tree P2P content distribution systems can be analytically derived, under two assumptions: (a) content is divided into very small pieces which flow through the trees as *fluid*, and (b) only *uplinks* in the physical network can be bottlenecks. By making the fluid approximation (a), each peer is essentially doing *cut-through forwarding* so there is no wastage of any bandwidth in the process. Assumption (b) is not unrealistic in practical network settings. For networks satisfying this assumption, the analysis is dramatically simplified. Mundinger coined the term *uplink sharing model* to refer to such networks [25]. Under these assumptions, it can be shown that the throughput limit R_{max} is bounded by

$$R_{max} \leq \min(U_0, \sum_{i=0}^{N} \frac{U_i}{N}) \tag{1.1}$$

where U_0 is the uplink bandwidth of the source, and $U_i, i > 0$, is the uplink bandwidth of each peer respectively [25, 13]. When there are constraints to the distribution trees, e.g. when the

peers do not form a full mesh, or when there are degree bounds to each node of the tree, the corresponding problem is addressed in [17, 21].

In practice, the performance of a multiple-tree P2P content distribution system depends on the *algorithms* used to implement tree construction and transmission scheduling. Such algorithms are distributed, involving the following subalgorithms at each peer:

1. find a set of neighbors, typically with the help of a *tracker* server and some randomization
2. determine a chunk to download, typically based on knowledge of neighbors' content and what is missing locally
3. request one or more neighbors for different pieces of the selected chunk

These algorithms have already been engineered in various P2P systems (such as BT, PPLive and others) through repeated experimentation and tuning. The question is, can we build reasonably simple models of these algorithms to help understand why they work and how to optimize them and make them more robust.

In the unstructured setting, the maximum throughput can be achieved, intuitively, if these two invariants can be maintained:

a) A peer's neighbors always have chunks the local peer needs
b) A peer's neighbors always have enough uplink bandwidth to satisfy the local peer's playback rate

Roughly speaking, the first invariant is achieved by *chunk selection*; whereas the second invariant achieved by *load balancing*.

The chunk selection algorithm has been studied in [1, 40, 29]. The key insight is to give sufficient priority to distribute the rare chunks which are those that have not propagated far along

the multiple trees yet. In the streaming case, chunks also have urgency, so the priority should take both rarity and urgency into consideration [40].

The objective of this thesis is to systematically study practical load balancing algorithms, that are decentralized, efficient, and easy to implement in real-life P2P systems. The load balancing problem has two special cases:

1. If all the neighbors can be used simultaneously, how to balance the load among neighbors to achieve the maximum utilization of their uplink capacity. This is called the *request allocation problem*, which is studied in Chapter 4.1.

2. If each peer is only allowed to select a small number of neighbors from its neighbor set to download simultaneously, which is usually the case in real system, how to find the best neighbors (with the most available uplink bandwidth) to help. This is referred as the *neighbor selection problem*. This case is studied in Chapter 4.2.

Shrewd readers will notice the similarity of our load balancing problem to that of multi-path congestion control. This is indeed the case, and we will draw the parallels and borrow ideas, as we discuss it in Chapter 2. There are, however, some important special features to our problem:

1. We are primarily interested in a streaming problem, where the objective is for all peers to achieve a throughput equal to a playback rate, R. The P2P file downloading case, where each peer tries to achieve the highest rate possible, is closer to the multi-path congestion control problem.

2. In our problem, a server is there to make up for any shortage of download rate, and the goal is to minimize the use of the server bandwidth. Theoretically, the server can offer its

service as a last resort. That is, when the load balancing algorithms can fully utilize peers' bandwidth, the bandwidth provided by the server is also minimized. However, this is easier said than done in a distributed environment where the use of server is determined by peers. How to make peers rely on the server only when necessary is discussed in Chapter 5.

3. In multi-path congestion control (and some rate control schemes e.g. [4]), it is important to consider *fairness*, which makes sure no host can dominate all the bandwidth. However, in our load balancing problem, we are only concerned with achieving maximum available throughput. Because peers who get lower bandwidth from others can directly ask help from the central server to fill in the gap, between their download rate and the playback rate, there is no need to require each peer shares the same downloading bandwidth.

Another distinction with the mostly theoretical study of multi-path congestion control is that we focus on practical algorithms, where discretization (removal of fluid assumption), and the location the controls are implemented (receiver rather than sender) must also be considered in the design.

The organization of the thesis is as follows: We first describe the abstract version of our problem in Chapter 2, to allow us to see what we are trying to achieve under idealized situations (centralized, fluid etc). Then we consider a number of distributed algorithms for the load balancing problem and study them using event-driven simulation, as described in Chapter 3 and 4. In Chapter 5, we consider the remaining challenge of how to adaptively minimize the server load, without harming peers' throughput. After a brief account for related works, we conclude in Chapter 7. By presenting them together, we hope to illustrate the insights we gained.

□ **End of chapter.**

Chapter 2

Abstract Model

In order to focus on the load balancing problem, let us assume each peer has all the content that any other peer may ever want (the first invariant already met). Both the *request allocation problem* and *neighbor selection problem* can be formulated abstractly as optimization problems in resource allocation.

2.1 Request allocation problem

As in Fig 2.1, let there be N peers, each acting as an uploader with an uplink capacity of $U_k, k = 1, \ldots, N$. Each peer, k, also acts as a downloader and seeks service from a randomly selected set of uploaders (other peers), S_k. S_k is referred to as the neighbor set of peer k, randomly assigned by a tracker. Denote the download rate obtained by peer k from the jth neighbor by r_{kj}. The goal for each peer is to make sure $\sum_{j \in S_k} r_{kj} = R$ where R is the *playback rate.*

The source serves as a back-up server (indexed by 0), assumed to have infinite capacity, $U_0 = \infty$. If a peer k cannot receive sufficient service rate from its neighbors S_k, then the back-up server steps in to fill in the gap. Denote the back-up server's service rate for peer k by r_{k0}. Combining this with the above

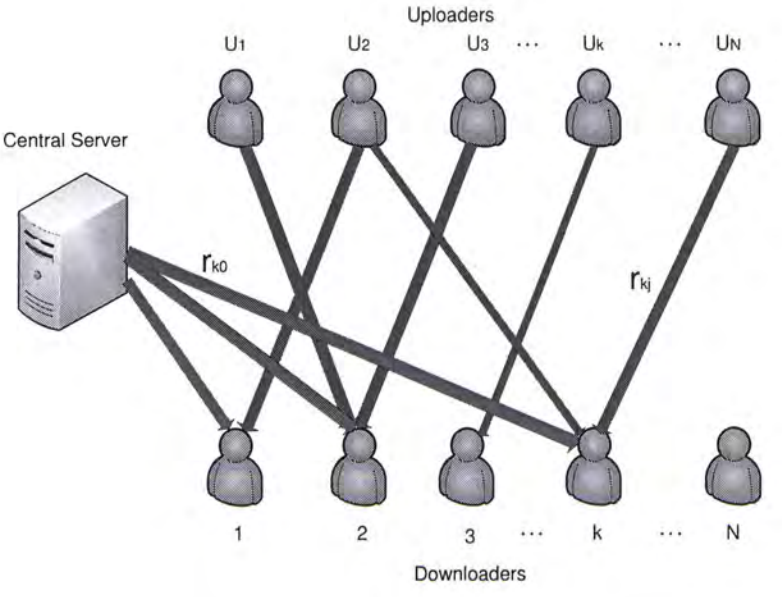


Figure 2.1: Load balancing problem

yield

$$r_{k0} + \sum_{j \in S_k} r_{kj} = R \quad \forall k$$

The objective is to select the service rates for each peer's neighbor set to minimize the total service rate of the back-up server,

$$r_0 = \sum_k r_{k0}$$

and the optimal solution is denoted as r_0^*. This is a standard *linear programming* problem.

By substituting r_{k0} with $R - \sum_{j \in S_k} r_{kj}$, it is easy to transform this problem into the following LP (2.3), which maximizes the sum of upload rate from peers. Theoretically, we assume that the central server can always fill in this rate gap, $R - \sum_{j \in S_k} r_{kj}$, exactly. However, in Chapter 5, it will be shown that it is non-trivial to design an algorithm that produces the rate from the server.

$$max \quad \sum_{k=1}^{N} \sum_{j \in S_k} r_{kj}$$

$$\begin{aligned} s.t. \quad & \sum_{j \in S_k} r_{kj} \leq R \quad \forall k \\ & \sum_{k:j \in S_k} r_{kj} \leq U_j \quad \forall j \\ & r_{kj} \geq 0 \quad \forall k, j \in S_k \end{aligned} \tag{2.1}$$

Since this is a standard linear programming problem, we know there exists a solution, and there are standard procedures for obtaining the solution (although it may take some time). Furthermore, we know that the optimal solution is bounded by the rate limit given by equation (1.1).

$$\begin{aligned} min \quad & \sum_{k=1}^{N} r_{k0} \\ s.t. \quad & \sum_{j \in S_k} r_{kj} + r_{k0} = R \quad \forall k \\ & \sum_{k:j \in S_k} r_{kj} \leq U_j \quad \forall j \\ & r_{kj} \geq 0, r_{k0} \geq 0 \quad \forall k, j \in S_k \end{aligned} \tag{2.2}$$

$$\begin{aligned} max \quad & \sum_{k=1}^{N} \sum_{j \in S_k} r_{kj} \\ s.t. \quad & \sum_{j \in S_k} r_{kj} \leq R \quad \forall k \\ & \sum_{k:j \in S_k} r_{kj} \leq U_j \quad \forall j \\ & r_{kj} \geq 0 \quad \forall k, j \in S_k \end{aligned} \tag{2.3}$$

From a practical point of view, our interest is in designing a distributed algorithm to find the optimal allocation, with good convergence time. In this respect, we can borrow many ideas from the congestion control literature where a variety of distributed algorithms (primal or dual) for solving similar resource allocation optimization problems have been studied. In simple terms, these algorithms can all be described by increase-decrease algorithms as in [2].

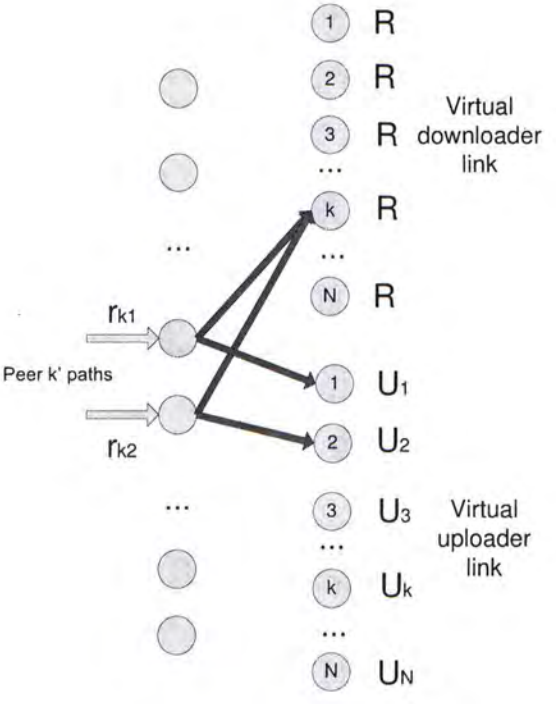


Figure 2.2: Mapping to congestion control problem

More formally, to map our problem to a classic congestion control framework, each overlay connection between peer k and its jth neighbor, $j \in S_k$, can be treated as a *path* that traverses two links, "virtual uploader link" with capacity U_j and "virtual downloader link" with capacity R. In Fig 2.2, we show an example of such mapping when peer k's neighbor set is $\{1, 2\}$.

Then the question is how to adjust the rate r_{kj} on path (k, j) to maximize the sum of all the rates without violating any capacity constraint on each virtual link. Let

$$R_k^d(t) = \sum_{j \in S_k} r_{kj}(t)$$
$$R_j^u(t) = \sum_{k: j \in S_k} r_{kj}(t)$$

be the total traffic on the downlink of peer k and uplink of

peer j, and $p_k^d(\cdot)$ and $p_j^u(\cdot)$ be the *price(s)* (or penalty functions) charged by the corresponding links. [11] showed that the rate adjustment algorithm given by the following differential equation would converge to a stable solution (that is the optimal solution of LP 2.3):

$$\frac{d}{dt}r_{kj}(t) = c(1 - p_k^d(R_k^d(t)) - p_j^u(R_j^u(t))) \tag{2.4}$$

where c is some constant step size.

However, the rate control algorithm in this form can not be directly implemented in practical P2P streaming systems. This is because in congestion control, each sender directly controls the sending rate r_{kj} on the jth path whereas in a pull-based P2P streaming system, each peer can not directly control the download rate from neighbors. This download rate is determined by the neighbor's uplink capacity and current load, which is usually unknown by the requesting peer. Instead of the rate, what peers can adjust is the number of pieces requested within one request message sent to neighbors. Such adjustments only affect the download rate from each neighbor indirectly. Therefore, rather than implement the rate control algorithm directly, we propose a request size adjustment algorithm to achieve load balancing for P2P streaming systems. The details will be discussed in Chapter 4.1.

2.2 Neighbor selection problem

Now, assume that peer k can only choose W_k, $W_k < |S_k|$, target neighbors from its neighbor set S_k and download from them simultaneously. This is quite likely the case in practice since peers cannot manage communication with too many neighbors at the same time. The question is how peer k should choose the W_k neighbors and adjust the rate at the same time. We

can extend the original problem definition as follows. Let λ_{kj} be a variable to indicate whether the jth neighbor is chosen by peer k. Then the optimization problem, including both request allocation and neighbor selection, becomes:

$$\begin{aligned} max \quad & \sum_{k=1}^{N} \sum_{j \in S_k} \lambda_{kj} r_{kj} \\ s.t. \quad & \sum_{j \in S_k} \lambda_{kj} r_{kj} \leq R \quad \forall k \\ & \sum_{k:j \in S_k} \lambda_{kj} r_{kj} \leq U_j \quad \forall j \\ & \sum_{j \in S_k} \lambda_{kj} = W_k \quad \forall k \\ & r_{kj} \geq 0 \quad \forall k, j \in S_k \\ & \lambda_{kj} = 0 \text{ or } 1 \quad \forall k, j \in S_k \end{aligned} \tag{2.5}$$

This is now a mixed integer programming problem. It is similar to the multi-path congestion control problem with dynamic routing, which is known to be more difficult to design a distributed algorithm for. [12] investigates the path selection algorithm of multi-path congestion control when only a few paths can be used simultaneously, which is similar to our case. They prove that if peers/users always change to a new path set with a higher sending rate, then such iterative path selection (without wrong moves) leads to the optimal path selection eventually. In practice, it is difficult to implement this algorithm because the performance of the new path set is unpredictable and there is no guarantee for a positive increment after each move. To support their algorithm, global information such as the available bandwidth and current loading at each neighbor needs to be available to avoid wrong decisions and synchronization among peers is also necessary to avoid oscillation, which is unlikely to be cost effective.

In Chapter 4.2, we propose and evaluate two kinds of heuristics that only depend on local information and are hence easy

to implement in real P2P streaming systems.

□ **End of chapter.**

Chapter 3

Simulation Model

The purpose of describing our problem at the abstract level, in the last chapter, is to first get a high level understanding of the problems we are trying to solve, and some generic algorithms under given assumptions, such as rate adjustment congestion control algorithms under fluid assumption. Our goal, however, is to develop some general algorithms without the fluid assumption, that are easy to implement in practical systems. In order to evaluate these practical algorithms, we develop a discrete-event simulation model that captures more details of a real system than the abstract model. Here, we first contrast our simulation with the Network Simulator 2 (NS2), which is a widely used discrete-event simulation platform targeted at networking research. After that, we focus on discussing the similarity and difference between the simulation model and the abstract model. The flow chart of this event driven simulation is shown in Fig 3.1.

Compared with the Network Simulator 2, our simulation model is simplified by three assumptions:

- In NS2, each layer of the networking architecture is simulated in detail. In contrast, our simulation model only focuses on the application layer behavior and the behaviors of lower layers are not taken into consideration, such

as routing dynamics, packet loss or congestion control.

- Our simulation model still follows the "uplink sharing" model, in which the network and peers' downlink can not be the bottleneck.
- In contrast to NS2, where it is possible to simulate the TCP/UDP packet transmission behavior, in our simulation model, only the chunk/piece level transmission behavior is considered.

Some basic assumptions are common to both the simulation and abstract models. The peer population is fixed at N, and each peer is randomly assigned a constant number of L neighbors by a tracker. All peers join the system at same time, but peers may have different uplink bandwidths. Each peer requests service from other peers as well as provides service when requested. How to request service and how to select appropriate peers are discussed in the next chapter. Each peer is assumed to have the content to serve any request. And finally, only the peers' uplinks can be the bottlenecks, which means the network can not be the bottleneck as in the uplink sharing model.

A major difference is to replace the rate-based control in the abstract model with something more like window-based control. Each chunk of video is divided into M pieces, and *piece* is the minimum unit of a request. When multiple requests arrive, they are queued and a peer serves them according to the FCFS principle as in standard queueing systems. The response time depends on the uplink bandwidth (service time) and the current queue length of the servicing peer (waiting time). Each requesting peer maintains a request window W, where $1 \leq W \leq L$. The request window is the maximum number of neighbors a peer can download from simultaneously. After all pieces requested in one request are downloaded, a peer can issue another request. Intuitively, larger M and W tend to help load balancing, making

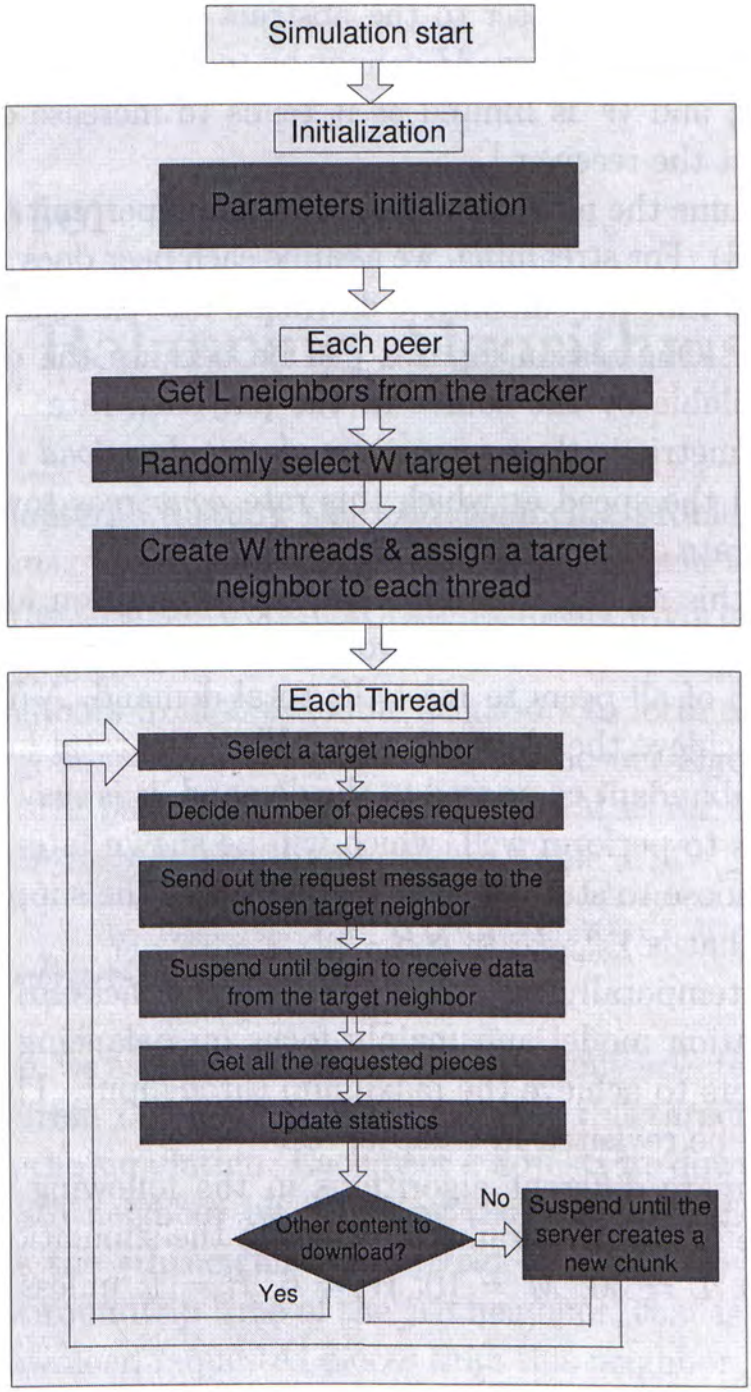


Figure 3.1: Flow chart of the simulation model

the discrete model closer to the abstract model with fluid approximation. In practice, M cannot be too large due to request overheads; and W is limited as it tends to increase error and loss rate at the receiver.[1].

We assume the playback rate is one chunk per unit time (e.g. one second). For streaming, we assume each peer does not make requests so that the cumulative download rate exceeds the playback rate. One reason for this can be because the content is made available by the source at the playback rate. The performance metric is the average *cumulative download rate* at all peers, and the speed at which this rate *converges* towards the playback rate over time.

Given this model, we know that a pre-condition for achieving (close to) playback rate throughput is for the total uplink bandwidth of all peers to meet the total demand, NR; namely all peers achieve the playback rate. When the total bandwidth supply is abundant compared to the demand, it is easy for many algorithms to perform well, which will be shown later. That is why we choose to study a tight regime where the supply equals demand, that is $\sum_{k=1}^{N} U_k \approx NR$.

Again, temporally, we exclude the source (back-up) server in the simulation model and mainly focus on balancing the load among peers to achieve the maximum throughput. The central server will be revisited in Chapter 5.

To compare different algorithms in the following chapters, we use the following parameter values in the simulation model: $N = 1000$, $L = 30$, $M = 10$, $W = 6$, $R = 1$; unless specified otherwise.

□ **End of chapter.**

[1]To deal with excessive incoming burst rate, we assume there is some lower layer pacing and error correcting/recovery mechanism.

Chapter 4

Load Balancing Algorithms

In this chapter, we study the load balancing problem as two subproblems: request allocation and neighbor selection. The first one tries to achieve optimal load balancing when each peer's W target neighbors are given, while the second tries to select W target neighbors from L available neighbors to form an optimal target neighbor set. In Chapter 5, we extend our algorithms to implement the back-up function by the central server. Notations used in this chapter are summarized in Table 4.1.

4.1 Request allocation

In this case, we assume that each peer randomly selects W target neighbors from L neighbors and sticks with this target neighbor set during the simulation. Each peer k adjusts its download rate from the jth neighbor by adjusting the number of pieces, s_{kj}, it requests the jth neighbor to upload. We use T_{kj} to denote the newest roundtrip time of the jth neighbor, that is, the total time to download requested pieces from this neighbor, including the propagation delay, transmission time as well as the queueing delay, shown in Fig 4.1. The average download rate of peer k in the previous second is R_k. As mentioned before, different from the congestion control problem where each sender can directly

Notation	Explanation
s_{kj}	Number of pieces requested by peer k from its jth neighbor
c	Constant step size to increase s_{kj}
T_{kj}	Download time of peer k from its jth neighbor
R_k	Peer k's average download rate in previous second
$\bar{r}_{kj}$	Peer k's download rate from its jth neighbor
Q_{kj}	Queuing time of peer k from its jth neighbor
τ_{kj}	Exponentially smoothed queuing time of peer k from its jth neighbor
V_{kj}	Variability of the queuing time of peer k from its jth neighbor
D_{kj}	Timeout threshold when peer k sends request to its jth neighbor
ρ	Retry ratio caused by timeout events
K	Number of neighbors replaced in one round
W	Request window size
$\hat{R}$	Download rate threshold used in section 4.3.3
$\bar{R}_k$	Peer k's average download rate in previous neighbor selection period

Table 4.1: Notations

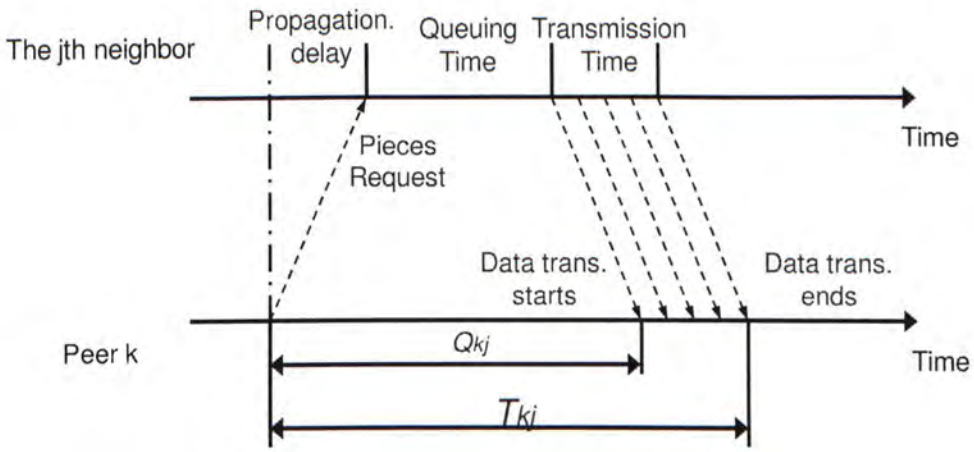


Figure 4.1: Download time of any request

control the send rate on each path, here peer k controls the number of requested pieces s_{kj} instead, which only affects the download rate indirectly. Following the rate control algorithm in Eq (2.4), we use the following rule to adjust s_{kj} in the $(n+1)$th iteration:

$$s_{kj}[n+1] = [s_{kj}[n] + c(1 - p_k^d[n] - p_j^u[n])]^+ \qquad (4.1)$$

where $p_k^d[n] = \frac{(R_k[n]/R - 1 + \epsilon)^+}{\epsilon^2}$, $p_j^u[n] = \frac{(T_{kj}[n] - 1 + \epsilon)^+}{\epsilon^2}$, and $\epsilon > 0$. Our *request size control algorithm* runs in the following way:

1. In the nth iteration, for peer k, if its jth neighbor is not overloaded, which means the service queue length of the jth neighbor, including peer k's request, is smaller than $U_j * 1s$, then the roundtrip time to finish peer k's request should be smaller than $1s$. Otherwise $T_{kj}[n] > 1$. Therefore, $T_{kj}[n]$, which is easy to measure locally, can be used as the feedback, indicating whether the jth neighbor is overloaded. If $T_{kj}[n] > 1$, the feedback induces peer k to reduce the number of pieces requested in the $(n+1)$th request message, sent to the same neighbor.

2. Similarly, whenever $R_k[n]$, the average download rate in the last second, exceeds the playback rate, $p_k^d[n]$ is positive. This causes peer k to decrease s_{kj} as well. In this situation, peer k already receive a large enough download rate, so it must not be in the way to prevent other peers achieving their playback rate.

3. If neither of the above two constraints are violated, s_{kj} is increased by c, a constant step size, in order to produce a larger download rate.

Here, ϵ is a small positive scalar which affects the convergence speed and the local stability of the algorithm. More detailed analysis of this factor can be found in [11] and [33]. Because the minimum requesting unit is a single piece, we also need to apply the randomization scheme to round s_{kj} to an integer number.

The obvious advantages of this algorithm include:

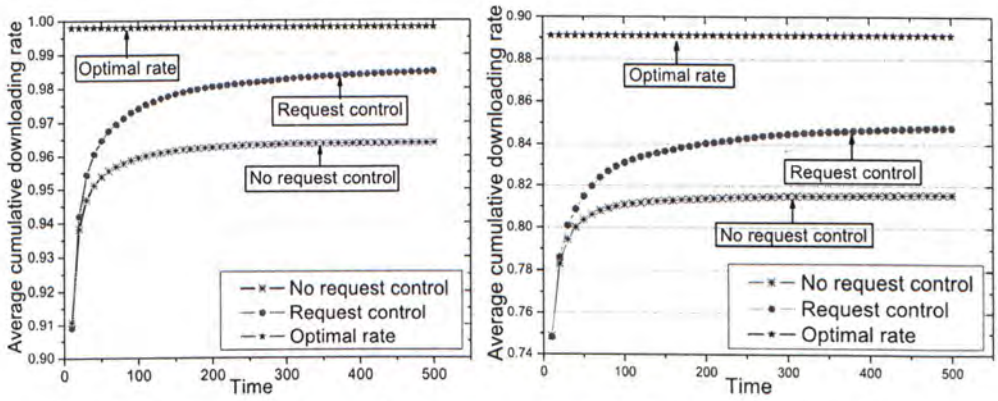
1. It only requires information that can be measured locally without incurring any extra information exchange in the network.

2. It runs in a totally asynchronous way. Not only is it unnecessary for peers to synchronize with each other, but for a particular peer, the request size to different neighbors can be adjusted asynchronously.

In our experiments, each peer initially sets s_{kj} to be 1 for all its neighbors. At the completion of each request, s_{kj} is updated according to the feedback T_{kj} and R_k and the algorithm (4.1). The default setting for c and ϵ is : $c = 0.1$ piece and $\epsilon = 0.1$.

In the first experiment, we assume a *homogeneous* network, by setting the uplink bandwidth of all peers to equal the playback rate. Meanwhile, one scheme is used as a benchmark, in which peers do not change the number of pieces requested, but

just stick to 1 piece per request. The optimal rate in this topology is obtained by solving the LP (2.3) in Matlab. The performance of the request size control algorithm, measured in average cumulative download rate, is shown in Fig 4.2(a). Obviously, the request size control algorithm performs better than the benchmark scheme, achieving an average download rate closer to the optimal rate. Even under asynchronous control, the convergence speed of this algorithm is acceptable, taking less than $25s$ to achieve 95% of its peak rate.



(a) Homogeneous network (b) Heterogeneous network

Figure 4.2: Average cumulative download rate of the request size control algorithm

From the figure, it can be observed that there exists a gap between the optimal rate and the rate achieved by our algorithm. Such a gap is unavoidable when we remove the fluid assumption, and when the algorithm runs in an asynchronous way. As shown in Fig 4.3, this gap decreases rapidly as the piece size becomes smaller, when the system gets closer to the fluid model.

We now motivate the importance of the neighbor selection algorithm. Without neighbor selection, each peer just randomly selects W target neighbors and sticks to them. Let us define the degree of peer k as the number of peers that choose peer k as the target neighbor, and show that such simple random scheme does not always work well because:

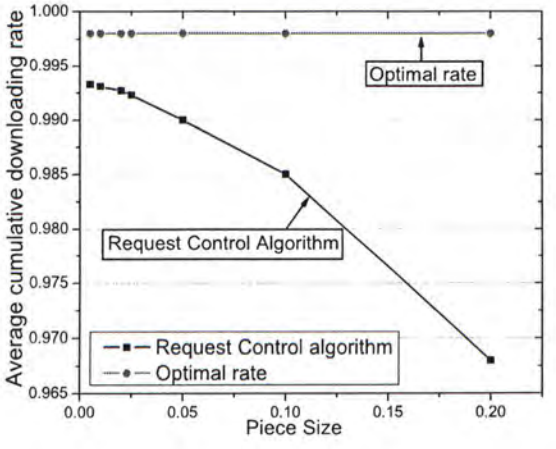


Figure 4.3: Performance gap between discrete model and fluid model

1. According to the analysis of the classic ball-and-bin model in [24], the probability that this simple randomization scheme will generate several peers with extremely high degree is almost 100% when the scale of the system increases. The uneven degrees make it hard to achieve load balancing.

2. In practice, the uplink capacity of peers can vary quite a lot, between ADSL versus Ethernet based access technologies. This randomization scheme also cannot guarantee that the degree of each peer is proportional to its uplink capacity. [20] provides a solid explanation of the reason.

To demonstrate the drawback of this static random neighbor scheme clearly, we construct a heterogeneous network, *Hetero_A* for short, in which there are equal number of three types of peers, with uplink bandwidths of (0.2,0.6,2.2) respectively, relative to the playback rate. In this setting, the result of the simple random scheme is shown in Fig 4.2(b). Unsurprisingly, compared to Fig 4.2(a), the performance deteriorates dramatically due to the heterogeneous setting and it is clear that such simple random scheme is incapable in the heterogeneous network setting. Two kinds of neighbor selection algorithms are studied in the

following sections.

4.2 Neighbor selection algorithms

Now, we allow peers to reselect their target neighbors and propose two kinds of algorithms: *timeout-based* neighbor selection algorithms and *periodic* neighbor selection algorithms. Both of them are easy to implement in practical P2P streaming systems by using the *multi-thread programming* technology.

Before introducing these two types of neighbor selection algorithms, the first issue is to identify an appropriate performance metric, used to evaluate neighbors' performance accurately and determine which one can provide better service.

4.2.1 What to measure?

In order to evaluate neighbors' performance accurately and cost effectively, a number of measurement options have been considered. For example, the uploader can indicate its queue length at the conclusion of uploading pieces of content. This information needs to be combined with the uplink bandwidth of this peer for it to reflect the load, but a peer does not automatically know its own uplink bandwidth. Alternatively, a central service can be used to request peers to query current load information or current available bandwidth. All these schemes seem to have their own problems; the major issue being the timeliness of the information versus the overhead of acquiring such information.

Without incurring extra information exchange, the scheme we pick for this study is based on measuring the download rate from the requesting peer. This scheme is relatively low cost and robust. For peer k, we use $\bar{r}_{kj}, j = 1, \ldots, L$, to denote the download rate from the jth neighbor. Each peer initially sets $\bar{r}_{kj}$ to be a large scalar for all neighbors, to make sure every

neighbor has the chance to be selected. After the completion of each request, the download rate from the jth neighbor is reset to the most recent measured value, namely the request size s_{kj} divided by the download time T_{kj}. Using $\bar{r}_{kj}$, peers are able to choose the suitable neighbor to download from.

4.2.2 Timeout-based neighbor selection algorithms

In this section, we introduce a simple neighbor selection algorithm of which the performance is not very good at first, and then add the *timeout and retry* mechanism to assist this kind of algorithm to achieve a better performance.

A simple neighbor selection algorithm

At the completion of each request, peers are allowed to choose the next target neighbor from the whole neighbor set, where the new request will be sent, rather than just keep using the initial target neighbor as in Section 4.1. More precisely, peer k, $k = 1, 2, ...N$, maintains W threads, each of which is in charge of one request independently. As illustrated in Fig 4.4, the algorithm executes as follows:

- *Step 0:* Initially, each thread of peer k is randomly assigned a target neighbor. Assume the target peer of the ith thread is the jth neighbor.
- *Step 1:* The ith thread generates a request and the number of pieces within this request is determined by Algorithm (4.1). Then, this request is sent to the jth neighbor and joins its service queue. Afterward, the ith thread enters the "PENDING" state, and waits to get the service.
- *Step 2:* When all the other requests ahead of this request are served, the jth neighbor starts to upload pieces to peer k and the state of the ith thread becomes "RECEIVING".

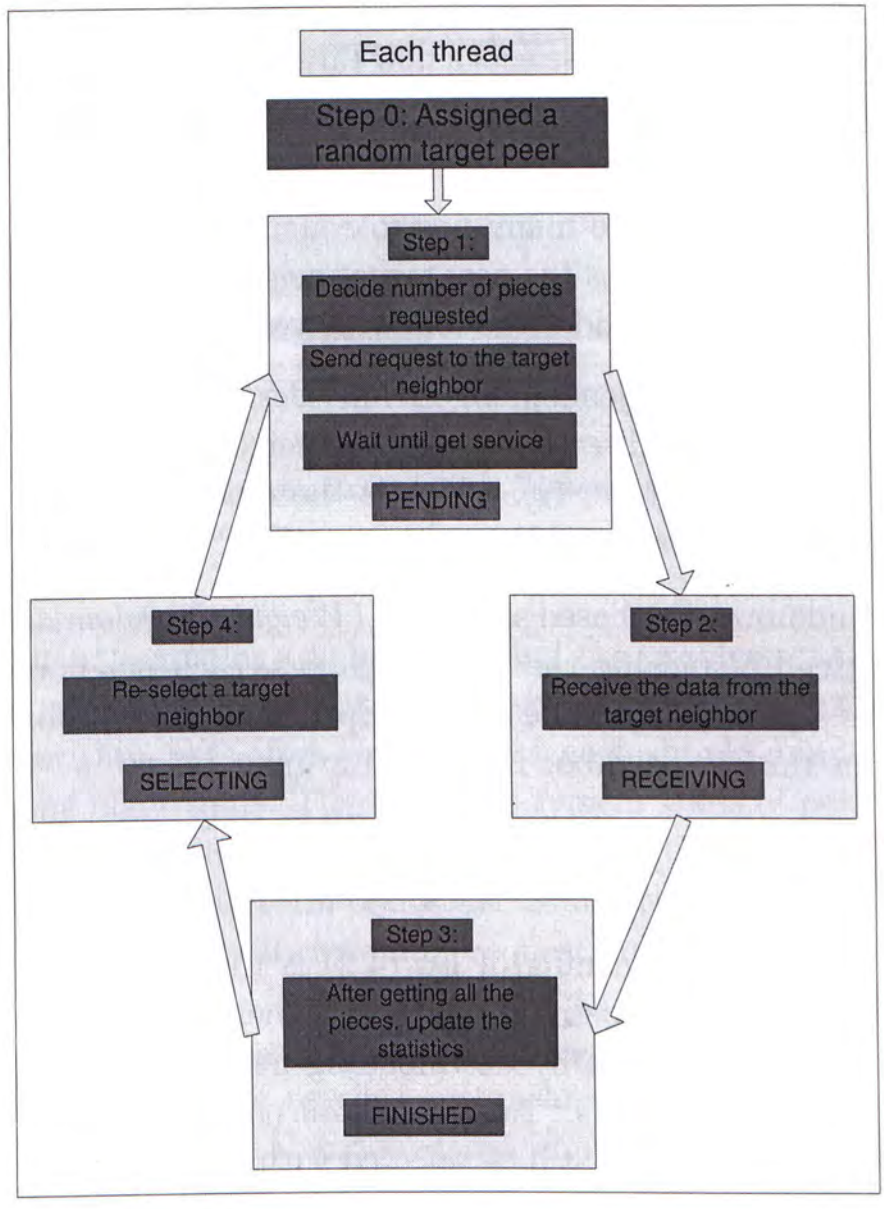


Figure 4.4: Flow chart of the simple neighbor selection algorithm

- *Step 3:* Once all the requested pieces are downloaded, the thread is in the "FINISHED" state and updates the download rate from the jth neighbor and other relevant statistics.

- *Step 4:* Based on the measured download rate of each neighbor, the ith thread selects a new target neighbor using some algorithms and goes to *Step 1* again. This state is "SELECTING".

In *Step 4*, there are many possible algorithms one can consider on how to chooses the new target neighbor from the whole neighbor set. We consider the following two algorithms:

1. Performance dependent algorithm (*Best Neighbor algorithm*): the ith thread directly chooses the neighbor with the highest download rate, $\bar{r}_{kj}$, which is the most straightforward method one can consider.

2. Randomization-based algorithm (*Weighted random algorithm*): Instead of picking the best neighbors, each neighbor now has a probability to be chosen. Specifically, the probability that the jth neighbor is selected is given below [1]:

$$p_j = \frac{\bar{r}_{kj}^2}{\sum_{j \in S_k} \bar{r}_{kj}^2}, j \in S_k \tag{4.2}$$

   The intuitive meaning of Eq (4.2) is to choose neighbors with high download rate with a higher probability; thus balancing the load by downloading more pieces from the less busy neighbors. Besides, those neighbors with small download rate will still be selected with a lower probability.

Note here, each thread makes its decision independently, without communicating with other threads.

We test their performance in the homogeneous network. However, as shown in Fig 4.5, these two algorithms do not work very well even in the homogeneous network, especially the best neighbor algorithm, achieving only 75% of the playback rate.

[1] The weight of the download rate from each neighbor is essential to deal with the heterogeneous case when peers have different uplink bandwidths. Purely choosing random neighbors for each request only yields worse performance.

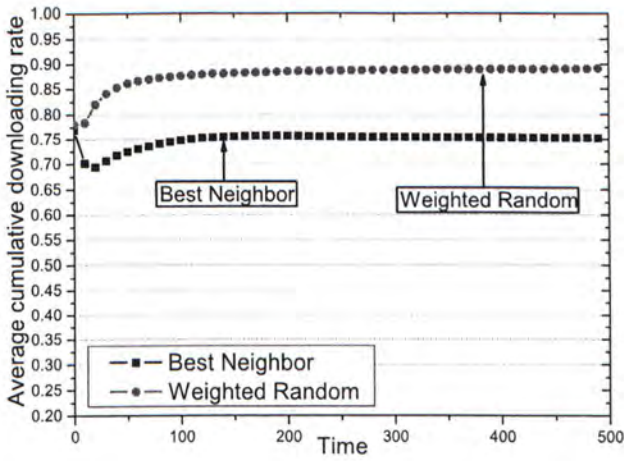
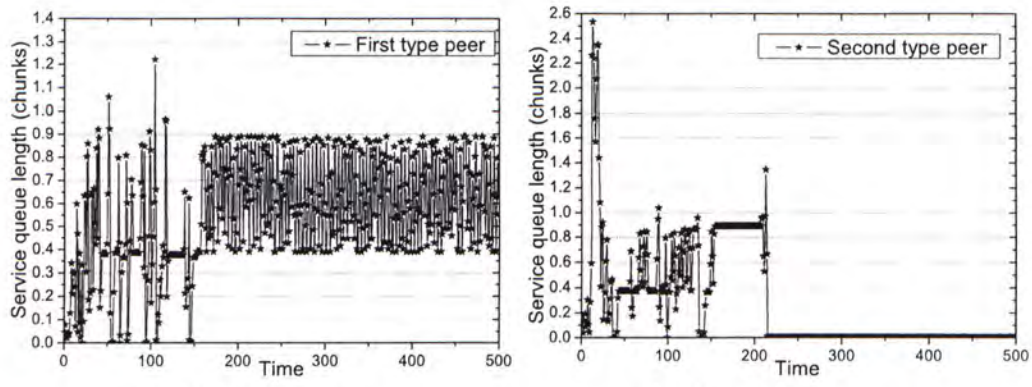


Figure 4.5: The performance comparison between the best neighbor and weighted random strategies

Without searching too hard, we find the problem of the best neighbor strategy. There is severe oscillation in the best neighbor algorithm, as is often the case with such simple-minded load balancing algorithms. There are two typical kinds of peers. For the first type, its incoming request queue length (load) oscillates wildly with time, in term of chunks, as illustrated in Fig 4.6(a). For the second type, its incoming request queue drops to zero after a while, as shown in Fig 4.6(b). In the latter case, the request queue has a sharp jump before dropping to zero, indicating the small download rate reached instantaneously is causing these type of peers to be abandoned by requesting peers. Roughly 25% of the peers are of the second type, explaining why only about 75% of the best possible throughput is achieved.

Another reason for the poor performance is that the estimated record of download rate may be outdated. Without incurring additional overhead in the network, $\bar{r}_{kj}$ is only updated when a download request sent to the jth neighbor is completed. Therefore, if the jth neighbor performs so poorly that peer k would never select this neighbor again, there is no chance $\bar{r}_{kj}$ will be updated anymore. Even if the service queue of the jth



(a) A first type peer (b) A second type peer

Figure 4.6: Average queue length of the best neighbor algorithm

neighbor is sometime empty sometime, this may not be noticed by peer k due to outdated load information. The second type of peer in Fig 4.6(b), abandoned by others, is exactly the case.

Compared with the best neighbor algorithm, the weighted random algorithm achieves better performance. The randomization scheme helps to some extent to avoid large oscillation, because not every peer directly selects the neighbor with the highest download rate. More importantly, neighbors with poor record still have a probability to be selected. This helps peers to discover those neighbors that are less loaded now, but performed poorly and were abandoned before. These two reasons explain why the weighted random strategy works better than the best neighbor strategy.

To further improve performance, one possible method is to require each peer periodically broadcast its load information to its neighbors. However, as we discussed before, such a method definitely increases the overhead and consumes some bandwidth in the network, which is not desired. Instead, we investigate a *timeout and retry* mechanism and combine it with our original algorithms together.

Timeout and retry mechanism

Originally, a timeout mechanism is built in to all algorithms to ensure we don't wait indefinitely for departed, or for extremely overloaded peers. Through more careful tuning of the timeout value, it is also possible to use this mechanism to avoid large oscillation and even directly shift load by retries.

The main idea is to estimate the queuing time for a request at each neighbor, assuming the system reaches steady state, and set a timer accordingly. If it takes significantly longer than the normal queueing time, one should give up and try sending the request to a different neighbor, because this indicates that the rate estimate for this target neighbor is outdated or the target neighbor is currently overloaded by many peers. Such a retry can effectively avoid large oscillation. Optionally, one can send a message to *Cancel* the original request. Otherwise, peers may receive duplicate data from multiple neighbors.

The queuing time at the jth neighbor, Q_{kj}, can be measured as the amount of time between when peer k sends out a request to the jth neighbor and when peer k receives the first data byte from that neighbor, shown in Fig 4.1 and updated at the beginning of *Step 2*. The exponentially smoothed version of the queueing time can be tracked as below:

$$\tau_{kj} = \theta * \tau_{kj} + (1 - \theta) * Q_{kj}, \quad 0 \leq \theta < 1, j = 1, 2, ...L \quad (4.3)$$

where Q_{kj} denotes the newest measured value and τ_{kj} the exponentially average queueing time.

In addition to tracking the average of Q_{kj}, it is also necessary to keep track of the variability of Q_{kj}, V_{kj}. It is computed as follows:

$$V_{kj} = \mu * V_{kj} + (1 - \mu) * |\tau_{kj} - Q_{kj}|, \quad 0 \leq \mu < 1 \quad (4.4)$$

When a request is sent to the jth neighbor, a timeout threshold,

D_{kj}, is set to:

$$D_{kj} = \tau_{kj} + C * V_{kj}, \quad C \geq 0 \tag{4.5}$$

The later term is used to deal with the possible variability of the queuing time as is done in transport protocols such as TCP.

Once the timeout mechanism detects longer than expected queueing delay, it considers this to be due to oscillation (rightly or wrongly), and retries the request with another peer. This comes at a cost - the requester sends two additional messages: one to cancel the original request, and the other to initiate the new request. The retry ratio, ρ, is defined as the number of timeout events divided by the total number of finished requests and measures the overhead of the timeout and retry mechanism.

When a timeout event occurs, besides canceling the request and selecting another neighbor for a new request, the local peer k also needs to update the download rate $\bar{r}_{kj}$ and queueing time τ_{kj}, because the old values are apparently inaccurate. Let C_1 and C_2 be the "penalty" factors and peer k updates $\bar{r}_{kj}$ and τ_{kj} in the following way:

$$\bar{r}_{kj} = \bar{r}_{kj}/C_1$$
$$\tau_{kj} = \tau_{kj} * C_2$$

In our experiments, the default value we use are: $\theta = 7/8, \mu = 3/4, C = 2, C_1 = 1.5, C_2 = 2$.

We add this timeout and retry mechanism to two strategies introduced previously: the best neighbor strategy and the weighted random strategy. The average cumulative download rate and the retry ratio ρ are shown in the Fig 4.7 and Table 4.2 respectively.

Obviously, compared with Fig 4.5, in steady state, the timeout and retry mechanism is able to improve the performance of the two strategies, while keeping the retry cost below 5%.

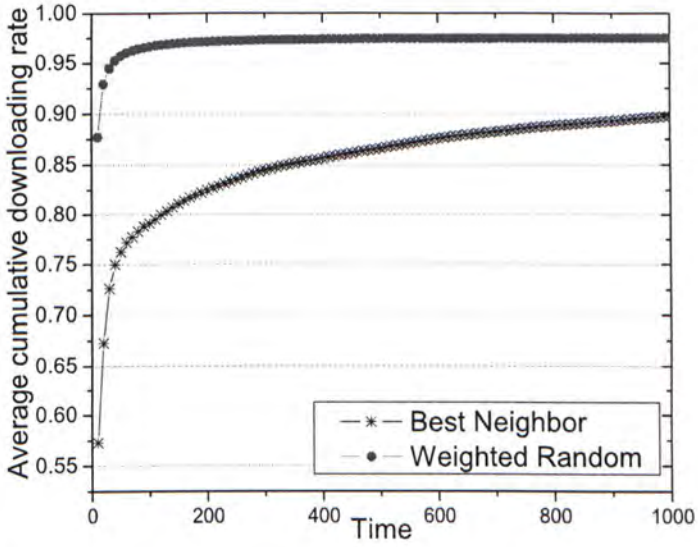


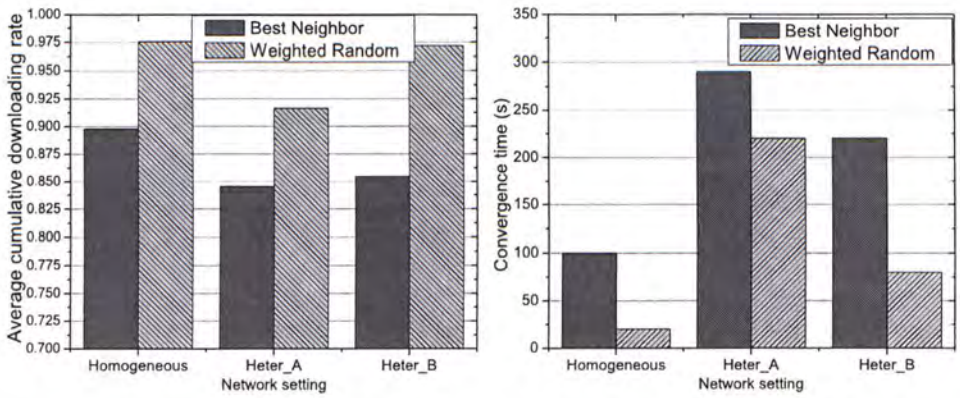
Figure 4.7: The performance of strategies with timeout mechanism

Table 4.2: The retry ratio of different algorithms under different network settings

Retry ratio: ρ			
	Homogeneous	Hetero_A	Hetero_B
Best Neighbor	0.0413	0.0508	0.0482
Weighted Random	0.0474	0.0472	0.0476

This validates our expectation. In addition, the weighted random strategy beats the best neighbor strategy again, achieving a larger average cumulative download rate, smaller retry ratio, and shorter convergence time, which is similar to the result without the timeout and retry mechanism.

In order to investigate the robustness of this timeout mechanism, we construct another heterogeneous network, *Hetero_B*, where the three types of peers have uplink bandwidths of (0.5, 1, 1.5). Because the shape of the curves for the different networks are similar as in Fig 4.7, we just show the peak value of the average cumulative download rate in Fig 4.8(a) and the convergence time, when the cumulative download rate achieved 95% of the



(a) Average cumulative download rate (b) Convergence time

Figure 4.8: Performance under different network settings

peak rate, in Fig 4.8(b). According to Fig 4.8(a), we observe that the best neighbor algorithm performs worse in the heterogeneous networks, which is because the peers with small uplink bandwidth are more likely to be abandoned by requesting peers. The weighted random algorithm works relatively stable in the different networks, higher than 90% playback rate. However, shown in Fig 4.8(b), the convergence time of the weighted random algorithm increases significantly in heterogeneous networks, which makes the weighted random algorithm less competitive if dynamic networks are considered.

4.2.3 Periodic neighbor selection algorithms

In this section, we propose another neighbor selection algorithm: *Periodic Neighbor Selection* algorithm, which is inspired by the periodic Tit-for-Tat neighbor selection algorithm used in BitTorrent. However, there are two important differences between BT and our problem:

- In BitTorrent, the TFT neighbor selection algorithm is implemented at the uploaders and allows an uploader to unchoke neighbors that contribute most. The downloader can

only download from neighbors that unchoke them. Therefore, from the view of downloaders, this can be treated as *passive* neighbor selection, because they do not have the power to select peers they want to download from. On the contrary, in most P2P streaming systems, it is assumed that peers do not limit their upload rate and are willing to serve requests from any peer. Therefore, in P2P streaming, our neighbor selection algorithm seems to be an *active* neighbor selection algorithm for the downloaders, allowing downloaders to actively select neighbors with high bandwidth to download.

- As addressed in [4], the TFT algorithm focuses on providing incentive to encourage peers to upload more, in order to be unchoked more times by others and download more. This is also referred to the *fairness*. Whereas, in P2P streaming, the most important thing is to guarantee peers' download rate, equal to the playback rate, rather than the fairness. How to provide the incentive to P2P streaming without harming system performance is still an open question. The major difficulty is how to encourage users with high uplink bandwidth to contribute more if their maximum download rate is only the playback rate as same as the rate at which the source generates the newest content.

Here, the timeout mechanism is only used as a lower layer mechanism to handle peers' leave and is not used by the periodic neighbor selection algorithm.

We allow peers to reselect their target neighbors periodically and the period is set to be 10s. Similarly, we still use the download rate, $\bar{r}_{kj}$, to determine which neighbor can provide better performance.

The basic idea of the periodic neighbor selection algorithm is that: At each decision moment, peer k sorts his current W target

neighbors according to their performance $\bar{r}_{kj}$. Then $K, 1 \leq K < W$, target neighbors with the worst performance are abandoned, and peer k chooses K new neighbors from the $L - W$ non-target neighbors, trying to get a better performance. Within each decision interval, our request size control algorithm (4.1) is used to fully utilize the uplink capacity of the target neighbors. The flow chart of this algorithm is similar to that in Fig 4.4 and the major difference is that, now each thread communicates with other threads and re-selects target neighbors periodically, rather than choosing a new target neighbor at the end of every request.

We propose three algorithms to choose K new neighbors from those non-target neighbors:

1. *Periodic Best Neighbor algorithm*, (P.B.N for short): peer k chooses the K neighbors with the highest download rate according to the rate $\bar{r}_{kj}$

2. *Periodic Random algorithm*, (P.R.D for short): since the number of neighbors L is more than the number of outstanding requests allowed, peer k randomly selects K new target neighbors without considering the past performance

3. *Periodic Weighted Random algorithm*, (P.W.R for short): Instead of selecting the best neighbors, each neighbor is chosen probabilistically . Specifically, the probability that the jth neighbor is selected is:

$$p_j = \frac{\bar{r}_{kj}^2}{\sum_{j \in S_k^-} \bar{r}_{kj}^2}, j \in S_k^- \tag{4.6}$$

   where S_k^- is the set of non-target neighbors.

The rationale behind these algorithms is that, searching for good neighbors increases the chance that some neighbors with

high uplink capacity, which were previously ignored are now selected. It is unavoidable that sometimes a peer makes a wrong decision; however it still keep $W - K$ neighbors with the best performance from the previous round and the loss in performance will not be a disaster. In the long run, the performance loss caused by poor decisions is expected to be minor compared to the potential gain from selecting better neighbors. The following experiment results validate our expectation.

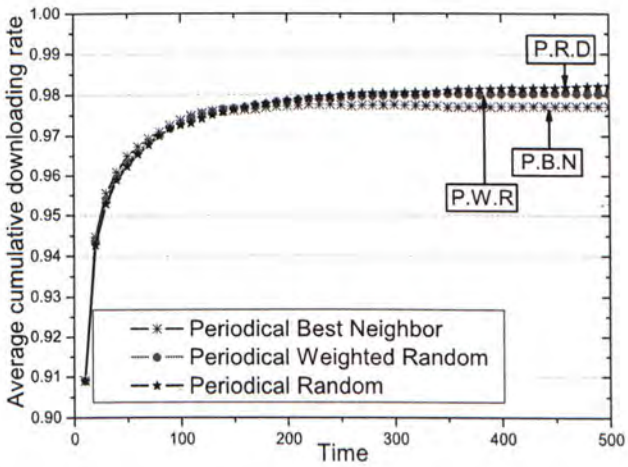
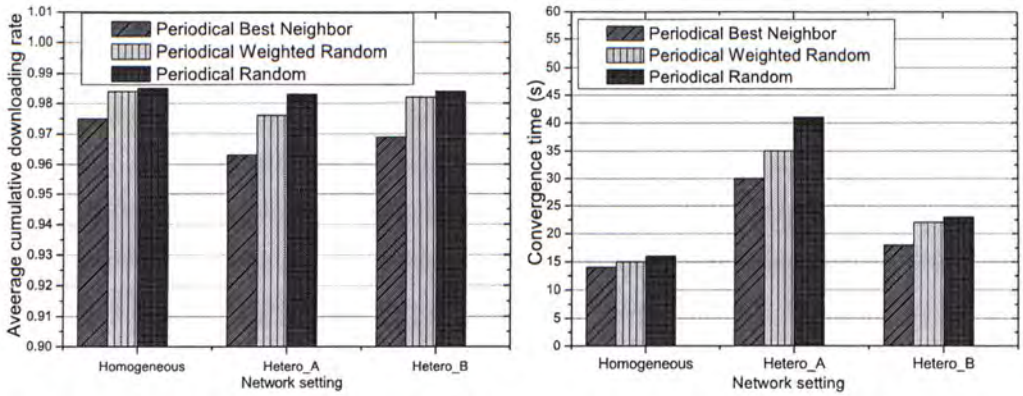


Figure 4.9: The performance of the periodical neighbor selection algorithms in homogeneous network

First, we test our algorithms in the *Homogeneous* network, and the average cumulative download rate is shown in Fig 4.9. It is noticed that all three algorithms perform very well, producing 98% of playback rate, which coincides with our expectation and the difference between them is not significant. One step further, to investigate the robustness of the periodical neighbor selection algorithms, we do the experiments in all three scenarios, *Homogeneous*, *Hetero_A* and *Hetero_B*. Their average cumulative download rate and convergence time is shown in Fig 4.10. The default setting of K is 2.

These three algorithms seem to have different strengths, and work reasonably well in different scenarios. For average cumula-



(a) The average cumulative download rate

(b) The convergence time

Figure 4.10: The performance of the periodical neighbor selection algorithms under different network settings

tive download rate, all three algorithms perform well in different scenarios, especially the random algorithm and the weighted random algorithm. Even though the best neighbor algorithm fluctuates the most, the different of its performance in different scenarios is still within 2% of the playback rate. This shows that they all achieve good robustness to deal with different network settings. However, a heterogeneous setting does have a strong effect on convergence time, especially for the random strategy. The reason is that it is harder to find those peers who still have available uplink bandwidth purely by random selection.

The random algorithm achieves the best average cumulative download rate while the best neighbor algorithm is the worst in all scenarios. The reason is evident when we randomly select one peer and record its degree change over time, for different algorithms using *hetero_A* (where the performance gap between different algorithms is the biggest), as shown in Fig 4.11. Similar to the timeout-based best neighbor strategy introduced in Section 4.2.2, there exists obvious oscillation when the periodic best neighbor strategy is used. One peer with a high uplink capacity may be noticed and chosen by many peers, then a large number of requests immediately overload this peer and each re-

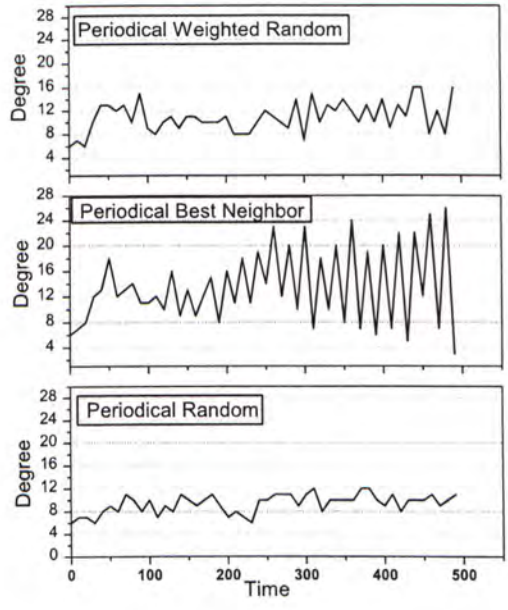


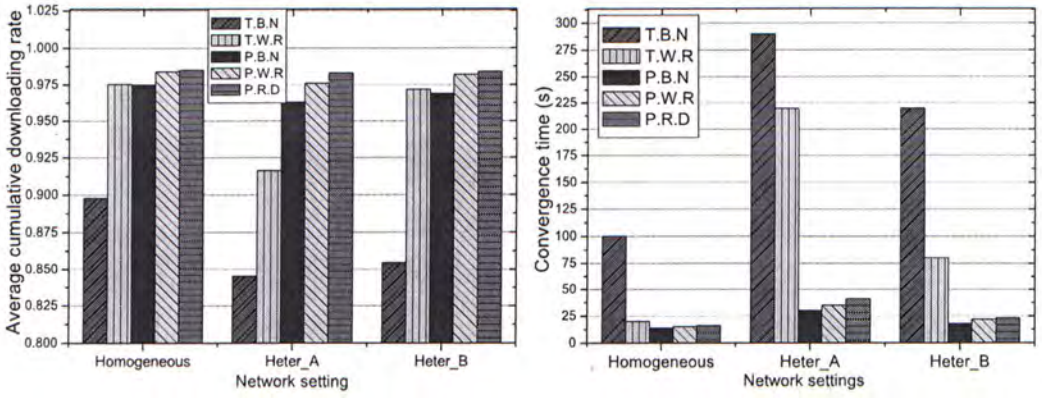
Figure 4.11: The oscillation of one peer's degree under different strategies

questing peer can only share a very small fraction of the uplink bandwidth. As a result, in the next round this peer will be eliminated by most of the neighbors and the degree of this peer drops sharply. Such oscillation occur repeatedly. Combining Fig 4.10(a) and Fig 4.11 together, it is clear that such oscillation has a critical impact on performance.

However, the performance and robustness of the random algorithm comes with a price, it has the largest convergence time. On the contrary, the best neighbor strategy can find and use peers with high uplink capacity immediately and this is the intrinsic advantage of load-dependent algorithms. Finally, as the combination of best neighbor strategy and random strategy, the weighted random strategy is always more moderate, achieving an acceptable average cumulative download rate and convergence speed.

4.2.4 Comparison: Timeout-based versus Periodical neighbor selection algorithms

In this section, we compare these two types of neighbor selection algorithm from three aspects, *implementation cost, overhead*, and *performance*. To be distinguishable, we add the prefix "P." to the periodic algorithms and "T." to the timeout-based algorithm.



(a) The average cumulative download rate

(b) The convergence time

Figure 4.12: Performance comparison: Timeout-based versus Periodical neighbor selection algorithm

- *Implementation cost*: First we compare these two algorithms from the view of implementation cost.
  - As introduced before, the timeout-based algorithm is convenient to implement using the multi-thread programming technology. Each thread is in charge of a single download request, as shown in Fig 4.4. In addition, each thread is independent of the others and whenever a thread re-selects a target neighbor, the decision can be made only according to the local record, without communicating with other threads.
  - On the other hand, to implement the periodical algorithm, the W threads need to communicate with each

other and work cooperatively. Such communication and cooperation help them to determine which threads keep the target neighbor unchanged and which threads eliminate the current target neighbors and choose new ones. In our simulation, to attain such cooperation, we add one extra "control" thread for each peer, which runs the neighbor selection algorithm periodically and informs the other W threads that which neighbor they should ask for pieces. Of course, there are other ways to implement the inter-thread communication, but this is beyond the scope of this thesis.

Therefore, without the inter-thread communication, the timeout-based algorithms is easier to implement in the software.

- *Overhead*: Only the timeout-based algorithm incurs extra message overhead within the network, because once a timeout event occurs, the peer needs to send out a *Cancel* message to the original target neighbor and then a new request to the new target neighbor.

- *Performance*: The comparison of the performance of these two kinds of algorithms is shown in Fig 4.12(a) and Fig 4.12(b). It is clear that the periodic algorithms achieve higher average cumulative download rate and shorter convergence time than the timeout-based algorithm. This is because of the intrinsic drawbacks of the timeout-based algorithm.

  1. Due to the lack of communication among W threads belonging to the same peer, multiple threads may select the same target neighbor to download and this actually decreases the size of the request window, which has a positive impact to the algorithms' performance. On the other hand, the periodic algorithm can always maintain

constant window size.

2. In addition, the periodic algorithm requires peers to maintain the $W-K$ target neighbors that provide highest download rate during last neighbor selection period unchanged. This reduces the impact of a poor decision. However, in the timeout-based algorithm, there is no such protection and each thread keeps selecting the new target neighbors. Therefore, wrong decisions will incur a big deterioration on the performance.

Accordingly, this provides a solid explanation why the timeout-based algorithm is beaten by the periodical algorithm.

In summary, the only advantage of the timeout-based algorithm is that it takes less effort to implement, but it is worse than the periodical algorithm in the other two aspects. Therefore, from now on, we only focus on the more competitive algorithm, the periodical algorithm, and try to tune it to get better performance by further experiments.

4.3 Further experiments

To understand these three periodic algorithms better and help to tune them to their best performance, we explore the impact of several parameters by further experiments. Besides, because of the robustness of our algorithms, results under different network settings are similar. Without confusion, we omit the word "Periodical" for short.

4.3.1 Request window size

The first important parameter is the request window size W. Increasing the request window size has at least the following two effects:

a) increasing the load in the network

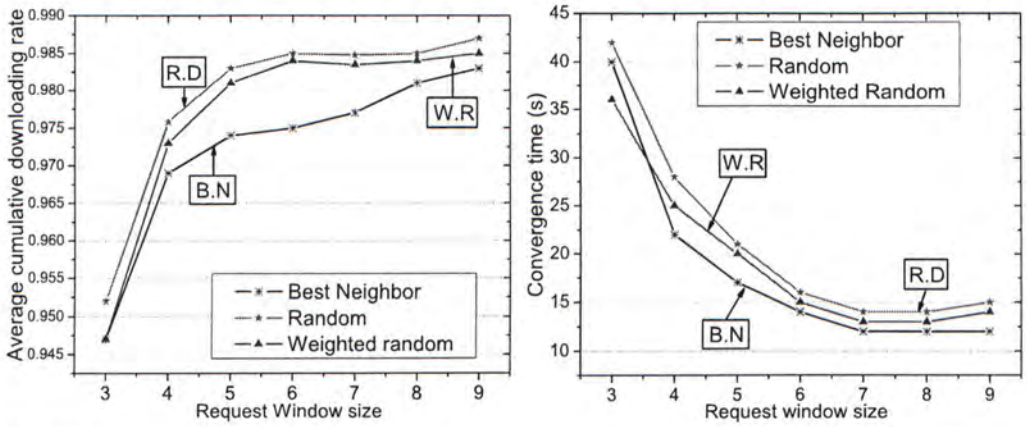
b) increasing the number of neighbors serving each peer

The first effect can be controlled - we can decrease the size of a piece at the same rate of increasing W so outstanding load does not increase. This way of maintaining a constant load is at the expense of some additional overheads. The second effect (of increasing W) has clear implication for load balancing.

As we increase W from 3 to 9, there is a noticeable improvement for all three algorithms, both in terms of average cumulative download rate and convergence speed, as shown in Fig 4.13, Fig 4.14 and Fig 4.15. By distributing load to more neighbors, a larger request window size tends to equalize the average uplink bandwidth of the neighbors requested by a peer, hence producing a more balanced load. The importance of the window size seems to be considered in the design of commercial P2P system. As mentioned in [9], the request window size is a critical parameter for performance; and in the PPLive VoD system, a relatively large window size is used to deal with the highly heterogeneous peer uplink bandwidth distribution. For a playback rate of around 500Kbps, they claim that 8-20 was the sweet spot and more than this number could still improve the achieved rate, but at the expense of heavy packet duplication rate. As shown in Fig 4.13, Fig 4.14 and Fig 4.15, this rule is also applicable in our P2P streaming model: a larger request window size is always preferable, unless the extra expense outweighs the performance gain.

4.3.2 Impact of K

The parameter K determines the number of neighbors to be replaced in each round. For a performance dependent algorithm, a larger K can reduce the convergence time because more potential neighbors with high uplink capacity can be selected in



(a) The average cumulative download rate

(b) The convergence time

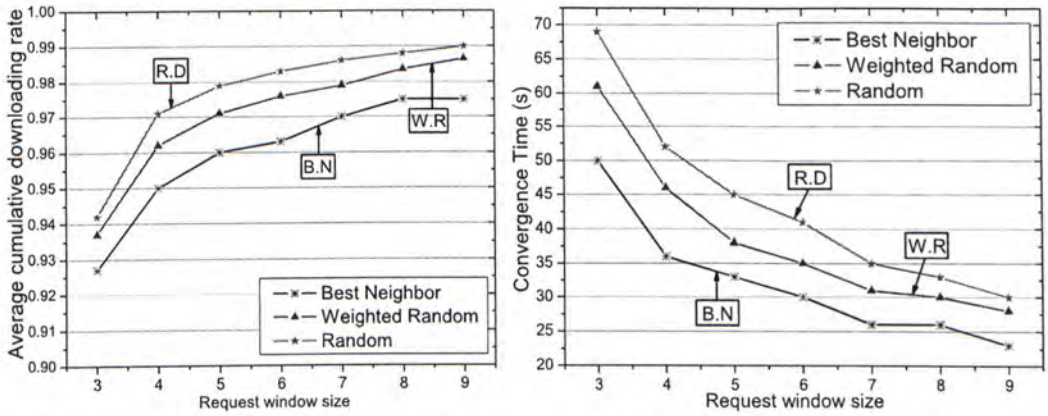
Figure 4.13: The performance under different W in the homogeneous network

one round of replacement. On the other hand, it also increases the possibility of oscillation, which plays a critical role in performance. Besides, with a large K, the impact of one wrong move may be significant because fewer neighbors with good performance used in last round are kept. The average cumulative download rate and convergence time under different K are shown in Fig 4.16, Fig 4.17 and Fig 4.18.

Performance of the random and weighted random strategies, they are unaffected by the value of K, because of the randomization. However, the convergence time of the best neighbor algorithm is negatively related to K. Since there is no way to resist oscillation, the average cumulative download rate also deteriorates with increasing K.

4.3.3 Adaptive adjustment of the neighbor selection period

The default setting of the neighbor selection period is 10s. However, we suspect that if this period can be adjusted intelligently, it is possible to improve the performance further. The reason is that if the download rate of one peer is already very close to the



(a) The average cumulative download rate

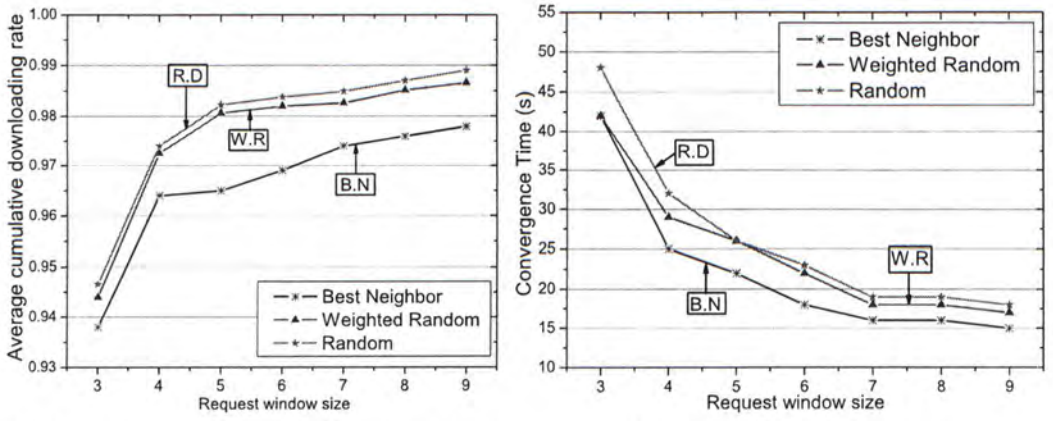
(b) The convergence time

Figure 4.14: The performance under different W in the heterogeneous network A

playback rate, then there is already no significant room left for improvement. By continuing to run the neighbor selection algorithm, this peer risks making a poor decision, which can harm performance. We therefore propose an adaptive method to deal with this problem. The basic idea is to decide whether to select new neighbors according to the average download rate during the previous neighbor selection period.

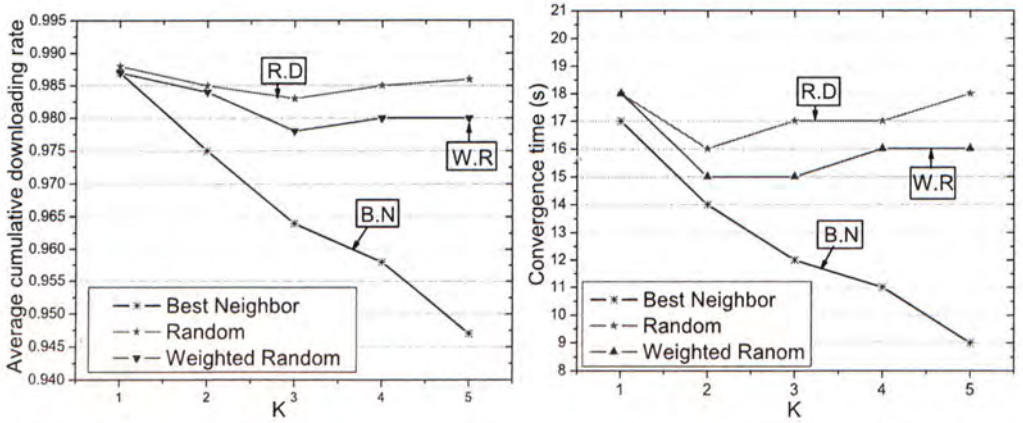
First, peers set a threshold $\hat{R}$ in advance and $\hat{R} = 0.9R$ in the following experiments. If the average download rate of peer k in previous period, denoted as $\overline{R}_k$, does not exceed $\hat{R}$, then this peer just runs the neighbor selection algorithm as usual. Once $\overline{R}_k > \hat{R}$, our adaptive algorithm steps into and a probability p is calculated as $p = \frac{\overline{R}_k - \hat{R}}{R - \hat{R}}$. With probability p, peer k just sticks to the current target neighbors without any change and with probability $1 - p$, the peer executes the neighbor selection algorithm. This prolongs the neighbor selection period of peers who already get high download rate.

We evaluate this algorithm for all three neighbor selection algorithms, shown in Fig 4.19. As expected, such a simple adaptive algorithm does provide an improvement in performance.



(a) The average cumulative download rate (b) The convergence time

Figure 4.15: The performance under different W in the heterogeneous network B

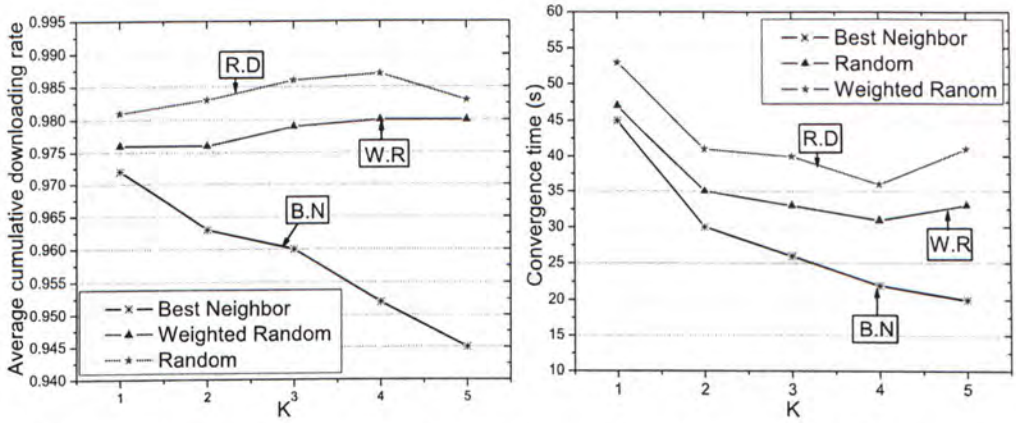


(a) The average cumulative download rate (b) The convergence time

Figure 4.16: The performance under different K in the homogeneous network

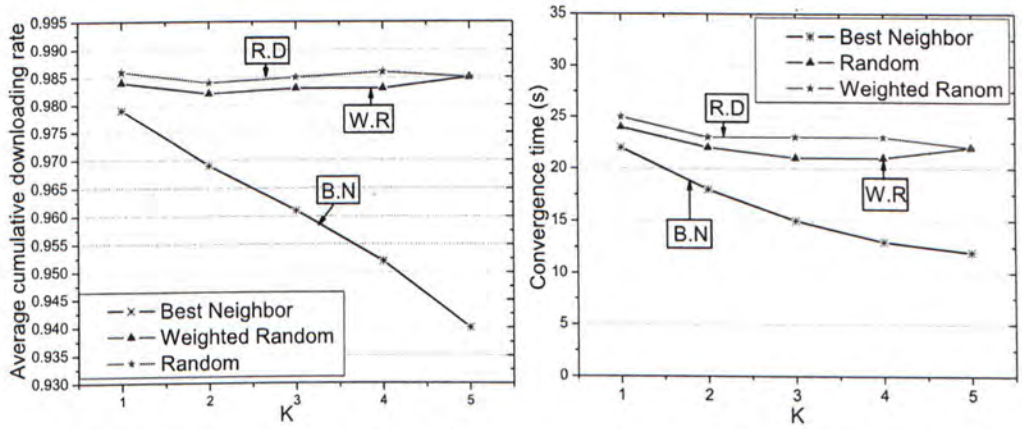
4.3.4 Performance with adequate bandwidth

In all the experiments so far, we considered only cases where the average uplink capacity equal the playback rate. This is deliberate to see how different algorithms fare under rather stressful situations. If the operator of the P2P network is willing to set the playback rate to a level below the average uplink capacity, we would expect all robust algorithms to achieve the playback rate on a cumulative basis. For the homogeneous network, we set the



(a) The average cumulative download rate (b) The convergence time

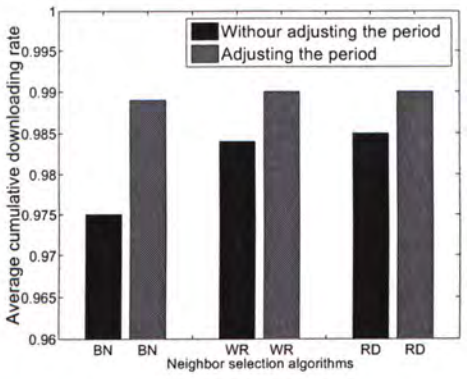
Figure 4.17: The performance under different K in the heterogenous network A



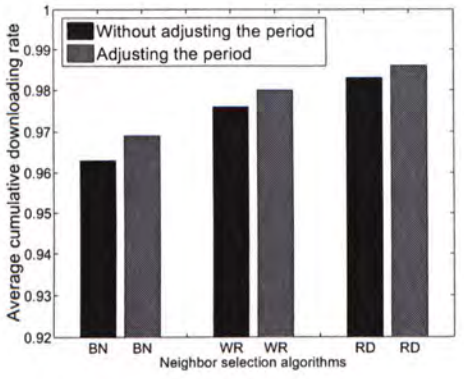
(a) The average cumulative download rate (b) The convergence time

Figure 4.18: The performance under different K in the heterogeneous network B

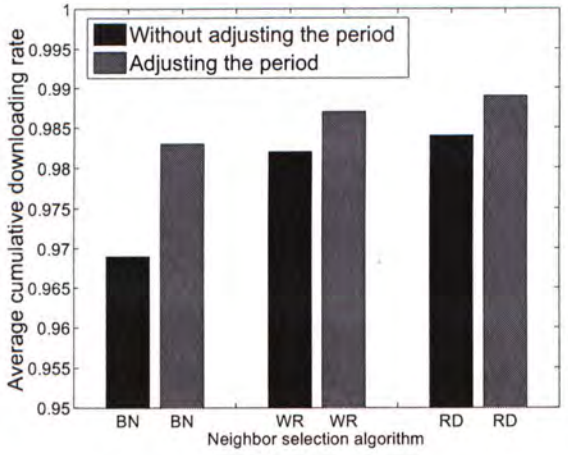
uplink capacity (of each peer) to $1.1R$; in the heterogeneous case A, we set the uplink capacity of the three types of peers to be $(0.22R, 0.66R, 2.42R)$ respectively (with an average of $1.1R$ as well). In the heterogeneous case B, the uplink bandwidth of the three types of peers is changed to $(0.55R, 1.1R, 1.65R)$ respectively. As shown in Fig 4.20, the average cumulative download rate of all algorithms is very close to the playback rate and the convergence time is below 30s. Besides, the difference between



(a) The homogeneous network



(b) The heterogenous network A

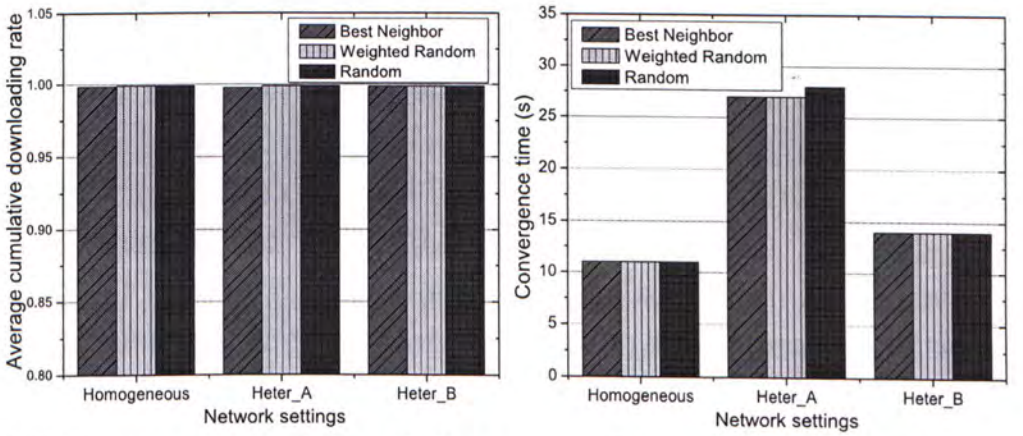


(c) The heterogenous network B

Figure 4.19: The average cumulative download rate of three algorithms with/without the adjustment of the neighbor selection period

different algorithms is very small. Therefore, if the system operates in the over-supply regime, all of the algorithms are quite competitive.

□ **End of chapter.**



(a) The average cumulative download rate

(b) The convergence time

Figure 4.20: The performance under networks with adequate bandwidth

Chapter 5

Minimizing Server's Load

In the previous chapter, we simply assume that the content server can magically fill in any gap between peers' download rate and the playback rate. More precisely, the download rate of peer k from the server is $r_{k0} = R - \sum_{j \in S_k} r_{kj}$. According to this assumption, once our load balancing algorithm fully utilizes the uplink bandwidth of neighbors and maximizes $\sum_{k=1}^{N} \sum_{j \in S_k} r_{kj}$, the total upload rate of the server is also minimized. However, in real systems, the questions of when to send a request to server, how many pieces to request, and how the server should response to all the incoming requests are non-trivial. Simply put, the central server faces the dilemma that, on one hand, the server can be so helpful that all peers' requests are always satisfied by

Notation	Explanation
α_k	Peer k's server load factor
$\tilde{R}$	Threshold used in algorithm 1
Δ_k	Step size to adjust α
β	Additive increase factor of Δ_k
γ	Multiplicative decrease factor of Δ_k
$\tilde{R}_k$	Peer k's average download rate in previous α adjustment period

Table 5.1: Notations

the server and thus peers will always download from the server and the uplink bandwidths of the peers become under-utilized, and on the other hand, the server tries to minimize its load and fails to provide sufficient guarantee for good peer playback performance. In this section, we study how the peers and the server can interact so that both parties are happy. The notations used in this chapter are listed in Table 5.1.

First, we assume the server responds to any peer request as follows: upon receiving a request from a peer, the server controls its upload rate and makes sure it takes exactly $1s$ to upload the requested pieces. If a peer requests too many pieces (e.g. larger than one chunk), the server simply uploads at the playback rate R. This constrain that the maximum upload rate from the server to each peer is R and guarantees that the server is only a back-up role[1].

Second, peers are only allowed to send requests to the server periodically. This is because if peers can send requests to the server at any time, it is very easy to overwhelm the server. In our experiments, the period is set to $1s$. Besides, to make sure the server is only treated as a back-up server, at the beginning of each second, peer k is allowed to ask for help from the server if and only if its download rate in the last second, denoted as R_k (that is the same as in the request size control algorithm (4.1), is less than the playback rate.

Given these constraints, as we show below, it is still challenging to determine how many pieces to request from the server. From the view point of peers, asking for more pieces is definitely a good choice, because the server is more stable than regular peers and can guarantee the Quality-of-Service (playback rate R). However, this places heavy load on the server and decreases the utilization of peers' uplink capacity.

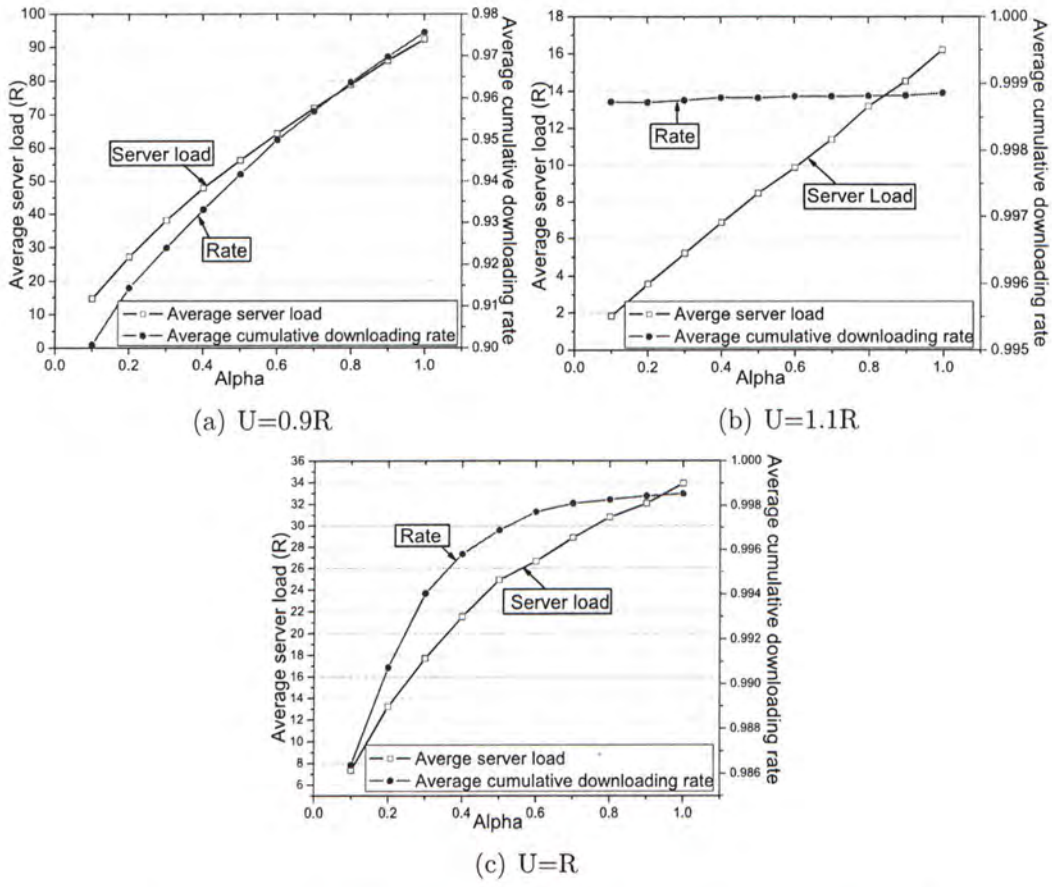
[1]If the dynamic system is considered, then it is possible that the server uploads content to the new arrival peers in a much higher rate, to minimize their start-up delay. This scheme is beyond the scope of this thesis.

We use a parameter α, $0 \leq \alpha \leq 1$, to help control the rate each peer can get from the server, as follows. If peer k asks the server for help (only when $R_k < R$), the number of pieces to be requested from the server is set to be $\alpha * (R - R_k) * 1s$, which is a fraction of the missing part downloaded in the last second. A randomization method is used to round any non-integral pieces. Intuitively, a large α increases the server load and guarantees a peer's download rate. However, a small α forces peers to try to get more from neighbors. Once their neighbors can provide more, the small α will not cause an obvious deterioration on the download rate but it decreases the server load.

To investigate the impact of α on peers' performance, the following experiments are carried out in the homogeneous network. Here peers are assumed to use the periodic weighted random strategy for neighbor selection. Similar result are also found when we consider other network settings and other neighbor selection algorithms. We first explore the impact of α under different uplink capacity setting exhaustively. The peers' performance and the average server load when the uplink capacity of each peer, U, is $0.9R$, $1.1R$ and R are shown in Fig 5.1.

In all scenarios, a larger α produces heavier server load, which is intuitively obvious. However, to the average cumulative download rate, the effect of α on the average cumulative download rate is totally different depending on which operating regime the system is in:

1. In the under-supply regime ($U = 0.9R$), a larger α increase the server load and also improves a peer's download rate significantly. This is because peers do not have enough uplink capacity to support the playback rate and the server must provide certain bandwidth. Therefore, in this regime, α should be close to 1 in consideration of peers' performance.



(a) U=0.9R

(b) U=1.1R

(c) U=R

Figure 5.1: The performance and server load under different α

2. In the over-supply regime ($U = 1.1R$), behavior is are totally different. A larger α does not bring significant improvement to peers' performance but only incurs heavier load on the server. Given a larger α, peers send more requests to the server rather than to neighbors that actually have adequate uplink bandwidth, and this decreases the utilization of peers' bandwidth. Therefore, for this regime where peers' uplink capacity is abundant, a small α is preferable to reduce the server load.

3. In the tight regime ($U = R$), a moderate value of α is suitable because when α is too large, the gain on performance is relatively small, compared with the increasing server load,

similar to the over-supply regime. Meanwhile, peers' download rate under small α are not quite competitive. It seems the suitable range is $\alpha \in (0.6, 0.8)$.

From this analysis, it is clear that α controls the tradeoff between peers' performance and the server load. At the same time, there is a tight relationship between α and peers' uplink capacity. An interesting question is whether there is any algorithm to adaptively adjust the value of α to suit different scenarios.

If the average uplink capacity of peers is known beforehand, maybe we can manually set α according to some mapping rules. But if we consider the dynamics of the system, such method can not follow the change of network conditions. Alternatively, we propose an adaptive, decentralized algorithm to adjust α periodically to a suitable value, which also takes the dynamics of peers' uplink bandwidth into consideration.

We use a threshold of the download rate, $\tilde{R}$, to prevent any serious deterioration of peers' download rate caused by the decrease of α. For peer k, $\bar{R}_k[n]$ is the average download rate during the nth adjustment period. Δ_k is the step size used to decrease or increase α_k, peer k's α. We used β and γ as the additive increase factor and multiplicative decrease factor of Δ_k, respectively. Besides, q is the upper bound threshold of the fractional performance difference after decreasing α_k. Our adaptive adjustment algorithm for peer k is shown in Algorithm 1.

In line 2 and 3, once peer k's average download rate is below $\tilde{R}$, α is increased immediately to get more help from the server. After each decrement of α_k, peer k requests fewer pieces from the server and tries to shift the load to neighbors to get equivalent download rate. If such decrement does not result in a relatively large drop on the download rate, larger than q, as in line 5, the step size is increased additively and α_k is decreased again, shown in line 6. Otherwise, α_k is reset to the previous value and the step size is reduced multiplicatively, as in line 8. The

Algorithm 1 Adaptive adjustment algorithm of α

Require: After the first adjustment period, record $\bar{R}_k[1]$ and then $\Delta_k = \Delta_0, \alpha_k = (\alpha_k - \Delta_k)^+$. Wait until the second adjustment moment.

1: **for** the nth adjustment moment, $n \geq 2$ **do**
2: **if** $\bar{R}_k[n] < \tilde{R}$ **then**
3: $\alpha_k = \min(1, \alpha_k + \Delta_k)$
4: **else**
5: **if** $\frac{\bar{R}_k[n-1] - \bar{R}_k[n]}{\bar{R}_k[n-1]} < q$ **then**
6: $\Delta_k = \Delta_k + \beta * \Delta_0, \alpha_k = (\alpha_k - \Delta_k)^+$
7: **else**
8: $\alpha_k = \min(1, \alpha_k + \Delta_k), \Delta_k = \Delta_k / \gamma$
9: **end if**
10: **end if**
11: Wait until the $n+1$ adjustment moment
12: **end for**

method to adjust the step size is inspired by the AIMD algorithm used in TCP congestion control. In the following experiment, $\tilde{R} = 0.95, \Delta_0 = 0.1, q = 3\%, \beta = 0.1, \gamma = 4/3$ and the length of adjustment period is $10s$.

To evaluate our adaptive algorithm, we take the dynamics of peers' uplink bandwidth into consideration. Initially, the uplink capacity of each peer is $1.1R$. Then after $500s$, it is changed to $0.9R$. After another $500s$, the uplink capacity is changed to R. The initial value of α is set to 1, which gives peers' performance the first priority.

The evolution of peers' average α, server load and the average cumulative download rate is shown in Fig 5.2. From $0s$ to $500s$, the system runs in the over-supply regime and our algorithm adaptively decreases α and the server load, without harming the performance because peers are able to find neighbors with enough bandwidth. From $500s$ to $1000s$, due to the decrease of uplink capacity, peers can not support each other and increase α immediately. The server steps in, and its load increases accordingly. α is decreased again after $1000s$, and more uplink bandwidth from peers are utilized and this immediately allevi-

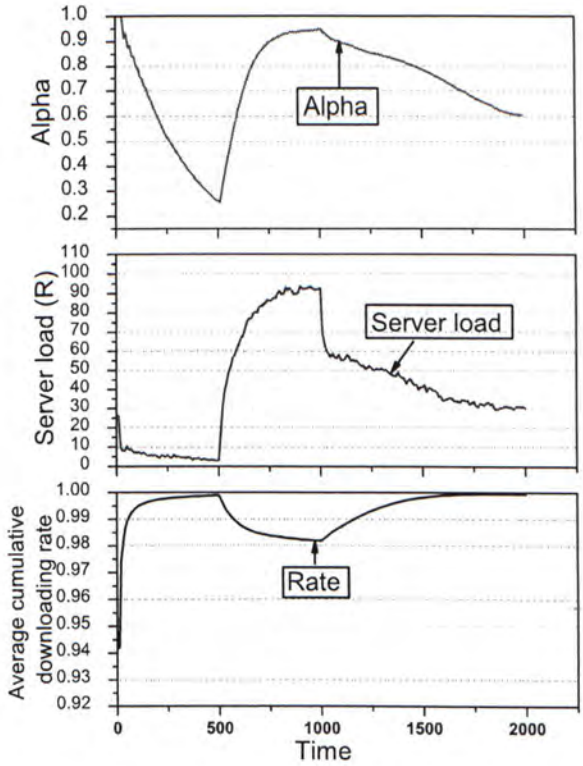


Figure 5.2: The dynamic evolution of α, server load and average cumulative download rate

ates the server loading. This shows that our adaptive algorithm can automatically sense the dynamics of network conditions and adjust α to a suitable value.

□ **End of chapter.**

Chapter 6

Background Study

6.1 P2P content distribution system

There is considerable interest in P2P content distribution system. In many ways, systems work seems to lead the way as there are many good examples of successfully deployed systems, such as [1, 39, 26, 27, 30, 31, 32], including the file sharing, streaming and Video-on-Demand systems. In trying to understand these experimental and commercial systems, there are considerable progress in theoretical analysis and measurement study.

6.1.1 P2P File sharing system

BitTorrent is the first widely deployed P2P file sharing system and attracts great interests from many researchers. In [25], based on the uplink sharing model, Mundinger et al proposed a deterministic scheduling algorithm to achieve a minimum makespan to download a file. [28] builds a simple deterministic fluid model to analyze BT's performance when the system is in the steady state. They show that such BT-like system has good stability and efficiency in the steady state. [23] extents this work by taking the stochastic behaviors into account and provides a detailed probabilistic model to analyze the stability and effectiveness of a P2P file swarming system. Meanwhile,

they also prove that the whole system throughout is asymptotically optimal by using the "random chunk selection" strategy. One step further, in [18], considering the uplink limit of each peer, they improve the result in [23] and derive a tighter performance bound. The last chunk problem, that it takes the longest time to download the last missing chunk, is also discussed in this paper. All these modeling papers give us a high-level understanding of why the BT-like system is scalable and effective. There are also several papers targeted at analyzing the special algorithms used in BT, such as chunk selection algorithm and peer selection algorithm.

The chunk selection strategy in BitTorrent is named *rarest first* strategy. Using this strategy, BT clients always prefer to download chunks which are the fewest of their own neighbor have first. As introduced in [3], this algorithms has two advantages:

- It efficiently increases the diversity of peers' available content. Such diversity makes sure peers have some chunks that others are interested in and leads a higher chance that peers can help each other.
- This also improves the availability of the whole file. In practice, peers may stop uploading at any time and this leads to a serious risk that some chunks are not available any more from all other peers. By focusing on the rarest chunks, such risk is reduced significantly.

Assuming a homogeneous closed network and random neighbor selection, Sujay et al [29] provide a simple model to analyze the performance of several chunk selection algorithms in the P2P file downloading system. The main result is that, any single side pull or push strategy is not optimal in disseminating all pieces to all users. Further more, a hybrid push and pull strategy is proposed and proved that it can achieve the near optimal download time.

On the other hand, the neighbor selection algorithm is also used in BitTorrent, referred as the Tit-for-Tat algorithm. According to the TFT algorithm, peers periodically unchoke four neighbors from which the download rate during last period is the highest. Besides, each peer also randomly selects one neighbor and is willing to upload to this neighbor, which is used to discover unnoticed neighbors with high uplink bandwidth. In [4], they provide an optimization framework to analyze the peer selection strategy and they claim that such TFT peer selection algorithm focuses on providing incentive to encourage peers to upload and eventually maximizing the fairness among peers. Quite differently, in P2P streaming system, peers usually can not manually limit their own upload rate and there is a tight playback rate constraint to satisfy. Assuming peers work cooperatively and do not limit the upload rate, our work focuses on the efficient peer selection algorithm which fully utilizes peers' uplink bandwidth and helps peers to catch up with the streaming constraint rather than providing fairness among peers.

There are also some experiment-based studies of BitTorrent systems. [15] validates three properties, which are proposed in theoretical study but have not been demonstrated clearly by experiments, including the clustering of similar-bandwidth peers, the effectiveness of BT's incentive mechanism and the high utilization of peers' uplink capacity. The behavior of the seeder is also first experimentally studied in this paper. In [14], a new uplink allocation algorithm is proposed to improve the performance of BT. The main idea is that, the uplink bandwidth of a peer should not be divided equally to several peers, because a chunk can not be downloaded under such low download rate. Their algorithm selects the unchoking neighbors by solving a *fractional knapsack problem*, and allocate the maximum upload rate to those neighbors. Through simulations, their new bandwidth allocation algorithm is validated to efficiently reduce the

download time by a factor of 2, compared with the original BitTorrent protocol.

6.1.2 P2P streaming system

The major difference between the streaming and file sharing is that the former has a playback rate requirement to satisfy, while the latter means best effort. In streaming, once the download rate is equal to the playback rate, the user experience is already very good. The higher download rate can not provide a significant improvement on user experience. On the contrary, in file downloading, the higher download rate is always preferred because the download time will be shorter. Therefore, different performance metrics are used to evaluate the performance of P2P streaming and file sharing systems. In streaming, the system performance is decided by whether most of peers can achieve the playback rate and watch the video continuously without glitches, whereas in file sharing system, in spite of the download rate of peers, the fairness is also a critical metric to make sure there is no free riders and everyone contributes to the whole system.

CoolStreaming [39] first designed a robust and efficient architecture for the P2P streaming system. The idea is similar with BitTorrent that every peer periodically exchanges data availability with several neighbors, and gets missing content from others or uploads content to others. Through experiments on the PlanetLab and Internet, it is shown that such simple data-driven architecture can successfully provide the streaming service to a large number of peers simultaneously and also achieve quite good video quality. In [16], they redesign the architecture to improve the performance further. The sub-stream technology is used in the new CoolStreaming and the chunk selection algorithms is modified to combine the pull and push strategy

together, which coincides with the analysis in [29].

Different from CoolStreaming, which designs the whole P2P streaming system from scratch, Bitos [34] implements the P2P streaming from BitTorrent protocol by minimum modifications. A video player is added into the BitTorrent software and the chunk selection strategy is altered to take both of the rarity and playback emergence into consideration. Theoretically, Zhou et al [40] established a probabilistic model of P2P streaming to evaluate and compare different chunk selection algorithms. Their analysis provided a solid explanation that why such mixed chunk selection strategy performs better in P2P streaming systems.

There are also considerable progress in developing theoretical model for the P2P streaming system. Based on the uplink sharing model, [13] established a stochastic fluid model for the full mesh network and analyzed the performance bound of the P2P streaming system. If peers does not form a full mesh or peers' uplink capacity is heterogenous, [21] and [20] derived the tight performance bound of the system. The key insight is that, if the degree of each peer is proportional to their uplink bandwidth, the system can still achieve the same theoretical performance bound as in a homogeneous network. However, the algorithms they proposed to achieve such bound is centralized, unscalable and can not be deployed in the real P2P streaming system. Different from their work, our work tries to design decentralized, scalable and implementable algorithms to actually improve the performance of real P2P streaming systems.

Measurement study of the commercial P2P streaming system also helps us to understand the system better. In [36], by analyzing the trace file of PPLive, the authors first are able to show that in real P2P streaming systems, there are some stable peers, which affect the performance of the overall system substantially. Secondly, they also proposed how to find those stable peers and organize them into an upper layer backbone network

to serve other normal peers. Hei et al [6] explores how to design a crawler and deploy passive sniffing nodes to collect the buffer bitmap of PPLive. Further more, they demonstrate that there is a tight correlation between the bitmap and peer's viewing-continuity and then the collected bitmaps can be exploited to monitor network-wide quality.

Based on data collected at the server side, Wu et al [37] have shown that the contribution of server still has a very important effect on the user experience and they proposed an online algorithms to adjust the server upload bandwidth among multiple channels to match the different forecast demand. Our thesis systematically addresses another important and generic component of a P2P streaming system: how peers adjust the rate they exchange content with each other (and the content server) to achieve load balance, to save content server load in P2P streaming.

6.1.3 P2P Video on Demand system

Recently, how to use the peer-to-peer technology to support video-on-demand systems is a new challenge in P2P area. Compared with streaming systems, peers in VoD system have less synchrony, which reduces the number of available content peers can exchange. Therefore, it is more difficult to alleviate the server load and maintain the user experience at the same time. To handle this, in the P2P VoD system, a small disk storage is contributed by every peer, usually 1 GB. This storage is used to replicate the viewed movie and in this way, peers can upload content viewed before to others, rather than just exchange current viewing data like in the P2P streaming system. This effectively increases the utilization of peers' uplink bandwidth and decreases the server load. How to efficiently replicate movies according to the system demand is also an interesting problem,

referred as the replication problem. [38] provides a preliminary study on this topic.

The feasibility and profit of the P2P VoD system is first discussed in [8] and [7]. Through modeling and simulation, they demonstrated that the P2P technology could dramatically reduce server bandwidth cost, especially if peers could actively pre-fetch content when there are spare bandwidth in the network. To avoid large amount of the inter-ISP traffic caused by the P2P architecture, some ISP-friendly strategies are proposed to localize the traffic with the ISPs.

PPLive and PPStream have already built and deployed the real P2P VoD system and millions of user have been served. Huang et al [9] conduct an in-depth study of the large-scale PPLive VoD system. Several important design issues and algorithms are introduced in that paper, such as content discovery, piece selection strategy, replication strategy and transmission strategy. Through the movie viewing record (MVR), they have demonstrated and analyzed several important properties of their system, e.g. user behavior, health index of movie, user satisfaction index and the server load.

6.2 Congestion control

As mentioned in the introduction, our load balancing problem has some similarity to the congestion control problem, as they both can be formulated as resource allocation optimization problems. Furthermore, there are important ideas from the congestion control literature about how to design distributed algorithms to solve such resource allocation problems. In the process of developing our algorithms, we discuss the parallels between these problems, use the relevant ideas from congestion control as a theoretical basis. Therefore, it is essential to provide a short summary on what progress have already been done in this area.

But more importantly, as discussed in this thesis before, we point out the differences between our problem and congestion control, and also carry the study further in studying discrete algorithms that can be easily implemented in real life systems.

The modern optimization framework of the congestion control is proposed by Kelly et al [11] and they formulate the congestion control problem as a utility maximization problem, shown in (6.1):

$$\begin{aligned} \max \quad & \sum_{i=1}^{N} U_i(x_i) \\ s.t. \quad & \sum_{i:l\in i} x_i \leq C_l, \quad \forall l \\ & x_i \geq 0 \end{aligned} \tag{6.1}$$

where $U_i(\cdot)$ is the utility function on the ith path. This problem tries to maximize the sum of utility among all paths without violating any capacity constraint of each link. By constructing a Lyapunov function, they claim that the maximum solution of this function is arbitrarily close to the optimal solution of (6.1). Further more, two kinds of decentralized algorithms, primal algorithm and dual algorithm, are proposed and their optimality and global stability is guaranteed in the absence of the propagation delay. The intuition behind their algorithms is that the sending rate should always keep increasing, but whenever there is congestion on any paths, the sending rate should be multiplicatively decreased to avoid serious congestion.

When the propagation delay can not be neglected, the algorithms may not always converge to the optimal solution. In [33], the authors investigated the decentralized sufficient condition to make sure the primal algorithm proposed in [11] can still converge.

In [22], the stability of the dual algorithm is analyzed. Rather than solve the problem (6.1) approximately, they claim that they can get the exact optimal solution of (6.1) by solving the dual

problem, shown in (6.2):

$$\begin{aligned} \min \quad & D(p) = \max_{x} \sum_{i=1}^{N} U_i(x_i) - \sum_{l} p_l(\sum_{i:l\in i} x_i - C_l) \\ s.t. \quad & p_l \geq 0, \quad \forall l \end{aligned} \tag{6.2}$$

The advantage of solving the dual problem is that now the problem is seperatable on each path. According to the projection theory, they designed a decentralized gradient projection method to solve problem (6.2) exactly. Meanwhile, they show that if the propagation delay on each path is bounded and the step size is extremely small, then the global stability of their algorithm still holds even when the propagation delay is not neglectable.

In [11], they already consider a more general optimization problem - multi-path congestion control, in which each end user can use several paths simultaneously, shown in (6.3)

$$\begin{aligned} \max \quad & \sum_{i=1}^{N} U_i(\sum_{s\in R(i)} x_s) \\ s.t. \quad & \sum_{i=1}^{N} \sum_{s\in R(i):l\in s} x_s \leq C_l, \quad \forall l \\ & x_s \geq 0 \end{aligned} \tag{6.3}$$

where $R(i)$ is the path/route set of user i. Both in [5] and [11], a primal algorithm is proposed, similar as the primal algorithm for the single path congestion control. Without the propagation delay, the global stability is guaranteed. However, as addressed in [5], when the propagation delay is considered, the sufficient condition, derived in [33] for the single path case, does not hold in the multi-path scenario anymore and one step further, the new sufficient condition for the multi-path case is derived. More importantly, they show that using multi-path controller can achieve a higher utility regime than that of using the single path controller.

Another interesting primal algorithm is designed and analyzed in [10]. Their main idea is that each link sends a binary feedback message to the users, indicating whether this link is congested. Users adjust the sending rate according to the number of congested links on each path, rather than the packet loss rate used in [5]. Given additional constraints on the step size and utility function, they get a stronger result than [5], that such adjust scheme can exactly converge to one of the optimal solutions, rather than an approximate solution.

The major difficulty of designing decentralized dual algorithms for the multi-path congestion control problem is that the utility function $U_i(\sum_{s\in R(i)} x_s)$ is not strictly concave. Consequently, the prime problem may have multiple optimal solutions and the dual problem may be non-differentiable. Besides, the dual algorithms also possibly causes severe oscillation in the network. Therefore, it is harder to solve the dual problem and avoid the oscillation.

There are two ways to handle this, *proximal method* and *sub-gradient method.* To deal with the lack of strict concavity, in [19], Lin et al add a quadratic term to the prime function, keeping the optimal solution unchanged. After this, they follow the standard proximal optimization algorithm to solve the modified problem and also construct the online algorithms which is much easier to implement. The convergence of their algorithms with/without measurement noise is also addressed in this paper. Based on the analysis in [35], when there is no propagation delay, it is proved that the sub-gradient method can solve the dual problem efficiently. However, if the propagation delay can not be ignored, they demonstrate that the sub-gradient method still causes the oscillation problem when the primal variables are recovered from the dual variables.

One important extension is the multi-path congestion control with dynamic routing, in which users have an available path set

but only can use several paths simultaneously. [12] is a pioneer in this area. Based on the classic ball and bin model, first they show that even in the worst case, the coordinated controller can achieve better performance than the uncoordinated controller. Afterwards, they show that, for both of the coordinated controller and uncoordinated controller, if users can always change to a new path set with higher sending rate, such iterative path selection will lead to the maximum social welfare eventually. This result provides a solid theoretical guideline when we design practical path selection algorithms. However, as discussed before, such path selection algorithm can not be directly implemented in the real applications, because the sending rate on the new path set is totally unpredictable. On the contrary, our neighbor selection algorithms can be directly deployed in the real P2P streaming systems.

□ **End of chapter.**

Chapter 7

Conclusion

The load balancing problem - how peers use distributed algorithms to simultaneously use multiple neighbors to help they achieve their downloading needs and minimize the content server's support - is an important and generic problem in P2P content distribution systems. It is the equivalent of the congestion control problem in the network and transport layers. In this thesis, we systematically analyzed the problem, pointed the important issues and studied a number of practical algorithms. In particular, we first studied how peers adjust their request sizes to meet their downloading needs; then considered how to adjust their neighbor set to optimize the system level performance for heterogeneous networks. Finally, we also studied how to use an adaptive algorithm to minimize the use of the content server.

There are many interesting directions for future studies. In this thesis, we focused on the P2P streaming case. In contrast, the P2P file downloading case may require some different algorithms since it tries to maximize throughput rather than minimizing server load. This is of great interest since it is needed for P2P VoD systems. Another angle is to extend this work for the dynamic population case.

Bibliography

[1] BitTorrent. http://www.bittorrent.com/protocol.html.

[2] D. Chiu and R. Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN Systems*, 17, 1989.

[3] B. Cohen. Incentives build robustness in bittorrent. In *Proceedings of the 1st Workshop on Economics of Peer-to-Peer Systems*, 2003.

[4] B. Fan, D. Chiu, and J. Lui. The delicate tradeoffs in bittorrent-like file sharing protocol design. In *Proc. IEEE ICNP*, 2006.

[5] H. Han, S. Shakkottai, C. V. Hollot, R. Srikant, and D. Towsley. Overlay tcp for multi-path routing and congestion control. In *ENS-INRIA ARC-TCP Workshop*, 2004.

[6] X. Hei, Y. Liu, and K. W. Ross. Inferring network-wide quality in p2p live streaming systems. *IEEE Journal on Selected Areas in Communications*, 25(9), Dec. 2007.

[7] C. Huang, J. Li, and K. W. Ross. Can internet video-on-demand be profitable? In *Proc. ACM SIGCOMM*, Aug. 2007.

[8] C. Huang, J. Li, and K. W. Ross. Peer-assisted vod: Making internet video distribution cheap. In *IPTPS*, Feb. 2007.

[9] Y. Huang, T. Fu, D. Chiu, J. Lui, and C. Huang. Challenges, design and analysis of a large-scale p2p vod system. In *Proc. ACM SIGCOMM*, Aug. 2008.

[10] K. Kar, S. Sarkar, and L. Tassiulas. Optimization based rate control for multipath session. In *Technical Report, TR 2001-1, CSHCN*, 2001.

[11] F. P. Kelly, A. Maulloo, and D. Tan. Rate control for communication networks: shadow prices, proportional fairness, and stability. *Journal of the Operationl Research Society*, 1998.

[12] P. Key, L. Massoulie, and D. Towsley. Path selection and multipath congestion control. In *Proc. IEEE INFOCOM*, 2007.

[13] R. Kumar, Y. Liu, and K. Ross. Stochastic fluid theory for p2p streaming systems. In *Proc. of IEEE INFOCOM*, May 2007.

[14] N. Laoutaris, D. Carra, and P. Michiardi. Uplink allocation beyond choke/unchoke or how to divide and conquer best. In *Proc. ACM CoNEXT*, 2008.

[15] A. Legout, N. Liogkas, E. Kohler, and L. Zhang. Clustering and sharing incentives in bittorrent systems. In *Proc. ACM SIGMETRICS*, pages 301–312, 2007.

[16] B. Li, S. Xie, Y. Qu, G. Keung, C. Lin, J. Liu, and X. Zhang. Inside the new coolstreaming: Principles, measurements and performance implications. In *Proc. of IEEE INFOCOM*, 2008.

[17] J. Li, P. A. Chou, and C. Zhang. Mutualcast: An efficient mechanism for content distribution in a peer-to-peer (p2p)

network. In *Proceedings of ACM SIGCOMM Asia Workshop*, Apr. 2005.

[18] M. Lin, B. Fan, J. Lui, and D. Chiu. Stochastic analysis of file swarming systems. *Performance Evaluation (Elsevier)*, 64:856–875, 2007.

[19] X. Lin and N. B. Shroff. Utility maximization for communication networks with multi-path routing. *IEEE Transactions on Automatic Control*, 51, May 2006.

[20] S. Liu, M. Chiang, S. Sengupta, J. Li, and P. Chou. Streaming capacity for heterogeneous users with degree bounds. In *Allerton Conference*, 2008.

[21] S. Liu, R. Zhang-Shen, W. Jiang, J. Rexford, and M. Chiang. Performance bounds for peer-assisted live streaming. In *Proc. ACM SIGMETRICS*, 2008.

[22] S. Low and D. E. Lapsley. Optimization flow control-1: basic algorithm and convergence. *IEEE/ACM Transaction on Networking*, 6.

[23] L. Massoulie and M. Vojnovic. Coupon replication systems. In *Proc. ACM SIGMETRICS*, pages 2–13, 2005.

[24] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12, Oct. 2001.

[25] J. Mundinger, R. Weber, and G. Weiss. Analysis of peer-to-peer file dissemination amongst users of different upload capacities. In *ACM SIGMETRICS Performance Evaluation Review*, volume 34, pages 5–6, 2006.

[26] PPLive. http://www.pplive.com/.

[27] PPStream. http://www.ppstream.com/.

[28] D. Qiu and R. Srikant. Modeling and performance analysis of bittorrent-like peer-to-peer networks. In *Proc. ACM SIGCOMM*, pages 367–378, 2004.

[29] S. Sanghavi, B. Hajek, and L. Massoulie. Gossiping with multiple messages. *IEEE Transactions on Information Theory*, 53:4640–4654, Dec. 2007.

[30] Sopcast. http://www.sopcast.com/.

[31] TVants. http://www.tvants.com/.

[32] UUSee. http://www.uusee.com/.

[33] G. Vinnicombe. On the stability of networks operating tcp-like congestion control. In *Proc. 15th IFAC World Congr. Automatic Control*, July 2002.

[34] A. Vlavianos, M. Iliofotou, and M. Faloutsos. Bitos: Enhancing bittorrent for supporting streaming applications. In *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, 2006.

[35] T. Voice. Stability of multi-path dual congestion control algorithms. *ACM International Conference Proceeding of the 1st international conference on Performance evaluation methodolgies and tools*, 180, 2006.

[36] F. Wang, J. Liu, and Y. Xiong. Stable peers: Existence, importance, and application in peer-to-peer live video streaming. In *Proc. IEEE INFOCOM*, 2008.

[37] C. Wu, B. Li, and S. Zhao. Multi-channel live p2p streaming: refocusing on servers. In *Proc. INFOCOM*, 2008.

[38] J. Wu and B. Li. Keep cache replacement simple in peer-assisted vod systems. In *Proc. INFOCOM*, 2009.

[39] X. Zhang, J. Liu, B. Li, and T. S. P. Yum. Coolstreaming/donet: A data-driven overlay network for efficient live media streaming. In *Proc. INFOCOM*, pages 2102–2111, Mar. 2005.

[40] Y. Zhou, D. Chiu, and J. Lui. A simple model for analyzing p2p streming protocols. In *Proc. IEEE ICNP*, 2007.