Analysis of Load Balancing Algorithms in P2P Streaming

Yongzhi Wang, Tom Z. J. Fu and Dah-Ming Chiu
Dept. of Information Engineering
The Chinese University of Hong Kong
{wyz107, zjfu6, dmchiu}@ie.cuhk.edu.hk

***Abstract*— In unstructured P2P content distribution systems, the most important algorithms to ensure optimal flow of content along multiple dynamically created distribution trees are piece selection algorithms and load balancing algorithms. This paper models practical load balancing algorithms and derive a number of insights.**

I. INTRODUCTION

Peer-to-peer (P2P) content distribution, whether streaming or file downloading, relies on one or more (explicit or implicit) distribution trees from the source to all the receivers (peers). By distributing the content via multiple trees, all peers can contribute to the process which is the secret of why P2P systems scale. Of course, this is based on the assumption that the bottleneck of the distribution system is at the peers' access links rather than in the transit network, a reasonable assumption up to certain scale.

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 that 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 paper, we focus on the P2P streaming problem.

Broadly speaking, there are two types of P2P systems: structured versus unstructured. These refer to the two different approaches 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 it belongs to. Since peers come and go, maintaining these trees becomes complicated. In the unstructured approach, trees are not built explicitly, but rather from an on-demand basis. 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 who 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 seem rather chaotic, it works incredulously well in many practical systems (such as BitTorrent [1] and PPLive [3]).

Theoretically, the throughput limit of such multiple-tree P2P content distribution systems can be established via fluid approximation. Namely, the pieces are assumed to be small so that each peer is doing cut-through forwarding, without wasting any bandwidth in the process. If all peers form a full mesh so that there is no constraint on what trees to build, the throughput must be limited by

$$R \leq \min(C_0, \sum_{i=0}^{n} \frac{C_i}{n})$$

where C_0 is the uplink bandwidth of the source, and $C_i, i > 0$, is the uplink bandwidth of each peer respectively [7], [8]. When there are constraints to the distribution trees, e.g. when the peers do not form a full mesh, or when there is degree bounds to each node of the tree, the problem is addressed in [15], [14].

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.

Intuitively, optimal throughput can be achieved if the subalgorithms can maintain these two invariants:

a) A peer's neighbors have chunks the local peer needs
b) A peer's neighbors 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], [12], [13]. The key insight is to give sufficient priority to distributing 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 be set taking both rarity and urgency into consideration [12].

The objective of this paper is to study the performance of the *load balancing* algorithm. This is the approach we take. We first abstractly formulate the problem, as a centralized optimization problem, which let us see the end result we are trying to achieve. This is done in section 2. Then we describe a number of distributed algorithms and study them using event-driven simulation, in section 3 and 4. Some of these algorithms are eventually abandoned; others are competitive under different situations. By presenting them together, we hope to illustrate the insights we gained. It is also our hope that this simulation based study, with the insights we gained, will lead to analytical derivations.

II. ABSTRACT MODEL

We formulate a more idealized problem than that described in the previous section in order to get a high level understanding of the problem we are dealing with. The key assumption is that the first invariance is already met; in fact, we assume each peer has all the content that any other peer may want. So the problem is purely how to balance the load among different peers. Under this assumption, the source (server) is not even necessary, other than serving the role of a back-up server when there is not even bandwidth among peers to help each other.

Let there be n peers, each acting as a server with a service rate of $R_i, i = 1, \ldots, N$. Each peer, k, seeks service from a randomly selected set of servers (other peers), S_k. S_k is referred to as the neighbors of peer k. Let the service obtained by k be denoted r_{kj} where $j \in S_k$. The goal for each peer is to make sure $\sum_{j \in S_k} r_{kj} = R$ where R is the *playback rate*. Without loss of generality, we assume $R = 1$.

The source serves as a back-up server (indexed by 0), assumed to have infinite capacity, $R_0 = \inf$. If a peer k cannot receive sufficient service rate from its neighbors S_k, then the back-up server steps in to fill the gap. Let the back-up server's service rate for peer k be denoted by r_{k0}. Combine this with the above, we have

$$r_{k0} + \sum_{j \in S_k} r_{kj} = 1 \qquad \forall k$$

And 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 r_0^*. This is a standard *linear programming* problem.

So from a theoretical point of view, we know there exists a solution, and there is a standard procedure to get the solution, although it may take some time. From a practical point of view, the interest would be on how to design a distributed algorithm to find the optimal; how to reduce the time it takes to converge to the solution, perhaps a good enough solution if it takes too long to get the optimal.

III. SIMULATION MODEL

We develop a discrete-event simulation model to study the algorithms, which captures more details than the abstract model in the previous section.

In this model, the peer population is fixed, at N peers[1], and each peer randomly selects a constant number of K neighbors. The peers may have different uplink bandwidths, which determine the service rates of the peers. Each peer requests service from other peers as well as provides service when requested. The request service time depends only on the uplink capacity of the servicing peer (which means the network is not the bottleneck. These are the same as the abstract model.

A major difference is that instead of rate-based control in the abstract model, the algorithms will deal with discrete pieces; so they can be thought of like window-based control, similar to congestion control algorithms in transport layer that we are familiar with. Each chunk of video is divided into M pieces, and *piece* is the unit of request. Each peer is assumed to have the content to serve any request. When multiple requests arrive, a peer serves them using the FCFS discipline as in standard queueing systems, the service time depending upon the uplink bandwidth of the servicing peer. Each requesting peer maintains a request window W, where $1 \leq W \leq M$. The request window is the number of simultaneously outstanding requests a peer makes. After a requested piece is downloaded, a peer can issue another request. Intuitively, larger M and W tend to help load balancing, making the discrete model closer to the model with fluid approximation. In practice, M cannot be too large due to request overheads; and W is also limited since it tends to increase error and loss rate at the receiver[2].

We assume the playback rate is one chunk per unit time (e.g. one second). For streaming, we assume each peer cannot make requests so that the cumulative downloading rate exceeds the playback rate, this is because we assume content is made available at the playback rate. For this reason, our performance metric is the average *cumulative downloading rate* at all peers, and the speed this rate *converges* towards the playback rate over time.

To reduce the complexity of the algorithms, we have not included the source (back-up) server in the simulation model[3]. If the cumulative rate cannot reach the playback rate, the difference is assumed to be taken care of by the back-up server. This assumption will not cause the throughput to exceed the playback rate either, since the request rate is limited to the playback rate.

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 achieving playback rate. When the total bandwidth supply is abundant compared to the demand, it

[1]Dynamic peer behavior will be considered in future work.

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

[3]When to send requests to this back-up server is non-trivial, and will be considered in future studies.

is easy for many algorithms to perform well, as we will see below. That's why we choose to study a regime when the supply equals demand, that is $\sum_1^N R_i = NR$.

When operating in a regime where total supply equals total demand, it is hard to avoid that at some peers the request rate is higher than the available bandwidth. So each algorithm needs some mechanisms to prevent instability (severe overloading). This can be done by setting a request queue threshold at the servicing peer, beyond which all new requests will be dropped. An alternative is to leave the responsibility with the requesting peer, by implementing a request timeout[4]. In practical systems, probably both mechanisms are implemented; but for the simulation model, we include a default request timeout mechanism depending on some multiple of the average round-trip time.

For the simulation study to compare different algorithms in the next section, we use the following parameter values: N=1000, K=30, M=8, W=4, Playback rate=1; unless specified otherwise.

IV. ALGORITHMS

A. Brief Summary

There are many possible algorithms one can consider. Ultimately, the algorithms we considered belong to two categories:

- load dependent algorithms
- randomized algorithms

In the former case, some mechanism is adopted for measurement load at the targeting peers and a selection is made based on load. In the latter case, since the number of neighbors K is more than the number of outstanding requests allowed, the M requests are sent to randomly selected peers. The conclusion is that both type of algorithms can be tuned to work. For the load dependent algorithms, it is important to avoid oscillation. For the random algorithms, it is necessary to deal with heterogeneous peers (with different uplink bandwidths) where a simple random strategy does not work.

B. Best neighbor strategy

Intuitively, to achieve load balancing each peer should get some sense of load at each neighbor and send the next request to that neighbor. This can be considered as some form of gradient method in solving the optimization problem described in the abstract model.

A number of load measurement options have been considered. For example, the servicing peer can indicate its queue size at the conclusion of uploading a piece of content. This information needs to be combined with the uplink bandwidth of this peer for it to reflect the loading, and a peer does not automatically know its own uplink bandwidth. Alternatively, there can be some central service for requesting peers to query current load information. All schemes for load measurement seem to have their own problem; the major issue being the timeliness of the information versus the overhead of acquiring such load information.

[4]Some form of request timeout is needed if peers can leave the system.

The scheme we picked for this study is based on measuring roundtrip time at the requesting peer. This scheme is relatively low cost and robust. For a given peer, we use $T_i, i = 1, 2, \ldots, K$ to denote the roundtrip time of the i^{th} neighbor, the total time to download a piece from this neighbor, including the propagation delay, transmission time as well as the queueing delay.

In our best neighbor strategy, each peer initially sets T_i to zero for all neighbors. After the completion of each request, the roundtrip time to that neighbor is reset of the measured value. The neighbor with the smallest roundtrip time is where the next request is sent, ties are broken by randomization. By trying all neighbors, it ensures no good neighbor will be missed.

In the first experiment, we assume a homogeneous network by setting the uplink bandwidth of all peers to be equal to the playback rate. We also set the propagation roundtrip delay to be 3% of the time to play one chunk. The performance of the best neighbor algorithm, measured in average cumulative rate is shown in Fig 1. At around 75% playback rate, the performance is obviously very poor.

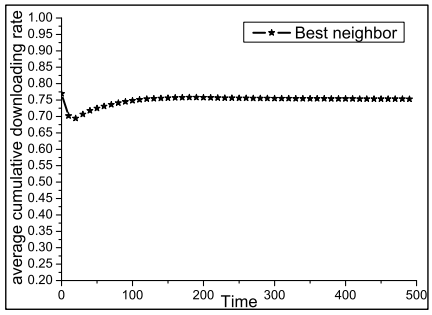

Fig. 1. Average cumulative downloading rate for the best neighbor strategy

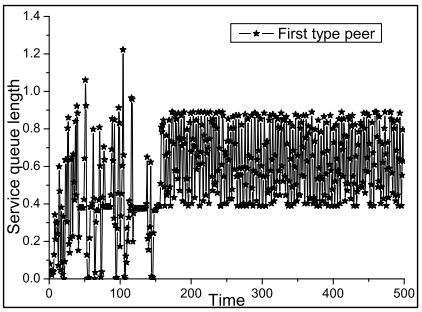

Fig. 2. The average queue length of a first type peer

Without searching too hard, we find the problem. There is severe oscillation, 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 oscillates wildly with time, as illustrated in Fig 2.

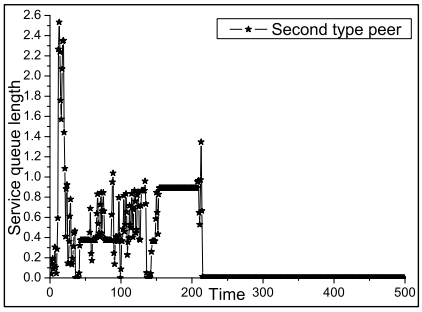

Fig. 3. The average queue length of a second type peer

For the second type, its incoming request queue drops to zero after awhile, as shown in Fig 3. In the latter case, the request queue has a sharp jump before dropping to zero, indicating the high roundtrip time 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 best possible throughput is achieved.

The next three strategies are chosen to overcome the oscillation problem.

C. Best neighbor with smoothed load measurement

To avoid the big spike in roundtrip time, hence the second type of wasted peers, one idea is to smooth the measured load values. Exponential averaging can be applied to roundtrip time in the obvious way:

$$\tau_i = \alpha * \tau_i + (1-\alpha) * T_i, \quad 0 \leqslant \alpha < 1 \tag{1}$$

where τ_i is the smoothed roundtrip time, whereas T_i is the latest measured value. The value α is used to tune the relative importance of the latest value compared to the average. When $\alpha = 0$, this is exactly the original best neighbor strategy; but when $\alpha > 0$, τ_i reflects an average performance of the ith neighbor rather than a one-shot performance measurement. We set $\alpha = 7/8$ in our experiments. The number of abandoned peers is reduced to about 10%, but oscillation is still quite severe. A peer, while not abandoned, might be not sought after for a long duration at a time. The improvement from the original Best Neighbor is quite marginal, as shown in Fig 4 together with some other strategies.

D. Random strategy

Another alternative is to remove load dependency, and try to balance load based on randomization. Each peer picks W neighbors randomly and sticks to these neighbors for service[5]. This simple strategy is surprisingly effective as shown also in Fig 4. As will be shown later, however, the simple random strategy runs into problems when the (heterogeneous) peers have different uplink bandwidths.

[5]Note, picking random peers for each request does not give better performance, but can yield worse performance in the heterogeneous case when peers have different uplink bandwidths.

E. Weighted random strategy

To prepare for heterogeneous peers, we modify the random strategy by applying weights to each neighbor according to its estimated load. Load estimation can be based on smoothed roundtrip time, as defined in Equation 1. Instead of picking the best neighbor, each neighbor now has a probability to be picked. Specifically, the probability neighbor i is selected is given below:

$$P_i = \frac{(\frac{1}{\tau_i})^2}{\sum_{j=1}^{K}(\frac{1}{\tau_j})^2}, \quad i = 1, 2, ...K \tag{2}$$

The intuitive meaning of Equation 2 is to choose neighbors with small τ_i with a higher probability; thus balancing the load by downloading more pieces from the less busy neighbors. Those neighbors with a large τ_i will still be selected with a lower probability. We simulated the weighted

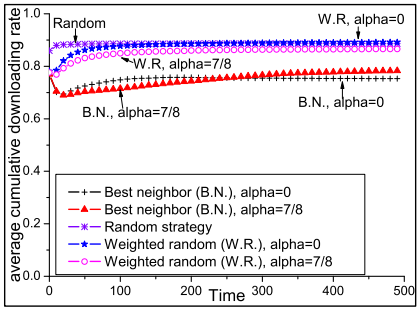

Fig. 4. The performance comparison between different strategies

random strategy with $\alpha = 0$ and $\alpha = 7/8$. The result is also shown in Fig 4 for comparison.

The performance of the weighted random strategy is similar to that of the random strategy, around 90% of the playback rate (Fig 4). But the purpose of the weighted random strategy, as will be shown later, is to deal with the heterogeneous case. Another property worth noting is that when $\alpha = 0$, the performance of weighted random is better than when $\alpha = 7/8$. This is the opposite of the result with the best neighbor strategy which performs better with $\alpha > 0$. When α is close to zero, the newly measured roundtrip time plays a more important role and, as discussed before, becomes a cause for oscillation. For the same reason, this less smoothed roundtrip time is also more helpful in detecting the less loaded neighbors. Since the weighted random strategy is already able to avoid oscillations through randomization, so it works better with the most currently measured roundtrip time (τ when $\alpha = 0$) to improve load balancing.

To help visualize the degree of balancing the load, we compute *the time to completion* for each peer, and plot the standard deviation of this quantity for each of the algorithms, see Fig 5. The time to completion of a peer is the waiting time for the last request in the current queue, and is a good measure of the current loading of a given peer. The smaller the standard deviation clearly indicates better load

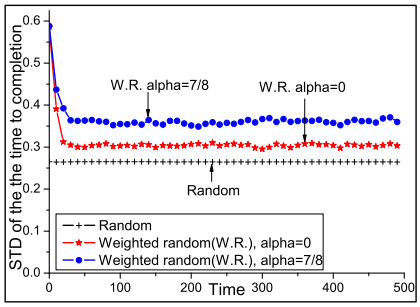

Fig. 5. The STD of the time to completion of three strategies

balancing. The random strategy has the lowest standard deviation, and it explains why it also achieved the best cumulative downloading rate.

V. FURTHER EXPERIMENTATION

In this section, we describe the results of further experimentation of the algorithms, to tune them for better performance and to study their robustness.

A. Timeout and retry

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 big 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 such a steady state, and set a timer accordingly. If it takes significantly longer than the normal queueing time, you should give up and try sending the request to a different neighbor. Optionally, you can send a message to *cancel* the original request.

The queuing time at the ith neighbor, Q_i, can be measured as the amount of time between when the peer sends out a request to a neighbor and when the peer receives the first data byte from that neighbor. The smoothed version of the queueing time can be tracked as below:

$$\theta_i = \beta * \theta_i + (1-\beta) * Q_i, \quad 0 \leqslant \beta < 1, i = 1, 2, ...K \quad (3)$$

where Q_i denotes the newest measured value and θ_i the exponentially average queueing time.

Besides tracking the average of Q_i, it is also necessary to keep track of the variability of Q_i, D_i. It is computed by the following equation:

$$D_i = \gamma * D_i + (1-\gamma) * |\theta_i - Q_i|, \quad 0 \leqslant \gamma < 1 \quad (4)$$

When a request is sent to the ith neighbor, a timeout threshold, T_i, is set to:

$$Z_i = \theta_i + C * D_i, \quad C \geqslant 0 \quad (5)$$

This is the same as how timers are set in transport protocols such as TCP.

When the timeout event occurs, besides canceling the request and selecting another neighbor for the request, the local peer also needs to update the roundtrip time τ_i and queueing time θ_i because old values are apparently underestimated. Let C_1 and C_2 be the factors for increasing the values of τ_i and θ_i respectively. In our experiments, the default value we use are: $\beta = 7/8, \gamma = 3/4, C = 2, C_1 = 1.5, C_2 = 2$.

Once the timeout mechanism detects longer than expected queueing, 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 pieces. The value of ρ measures the overhead of the timeout and retry mechanism.

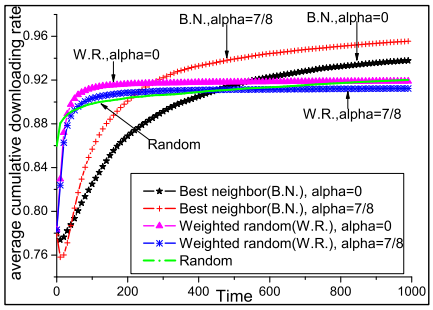

Fig. 6. The performance of strategies with timeout mechanism

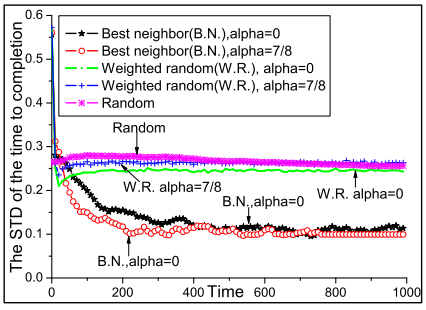

Fig. 7. The STD of the time to completion of strategies with timeout mechanism

We add this timeout and retry mechanism to all the strategies introduced in the previous section: the best neighbor strategy with $\alpha = 0$ and $\alpha = 7/8$, the random strategy, and the weighted random strategy with $\alpha = 0$ and $\alpha = 7/8$. The average cumulative downloading rate, the completion time STD and the retry ratio ρ are shown in the Fig 6, Fig 7 and Table I respectively. In steady state, the timeout and rety mechanism is able to improve the performance of all the strategies, while keeping the retry cost below

TABLE I

THE RETRY RATIO OF DIFFERENT STRATEGIES

Retry ratio: ρ				
B.N.		W.R.		Random
α=0	α=7/8	α=0	α=7/8	-
0.0559	0.0284	0.093	0.0859	0.0086

10%. The best neighbor strategy (especially with $\alpha = 7/8$) makes the most improvement, achieving nearly 95% of the playback rate, and the lowest completion time STD, of around 0.1. In comparison, the random strategies do not see as much difference. The reason is because the best neighbor strategy had more overloaded peers due to oscillations and the more aggressive timeout and retry strategy is able to correct that problem. Nonetheless, the convergence speed of Best Neighbor is rather slow. Another observation is that the retry ratio ρ always increases when $\alpha = 0$ compared to when $\alpha = 7/8$. This is because any strategy relying on the instantaneous load information (roundtrip time) always tends to generate more load variations (oscillations), hence results in more timeouts.

In the timeout mechanism, it is important to choose the right timeout threshold. If this threshold is too large, then the mechanism does not avoid long queueing times. On the other hand, if it is too small, each peer becomes impatient and the retry ratio increases, incurring more overheads. Recall that the timeout threshold is computed using Equation 5, in which the first part is the approximate queuing time and the second part is the approximate variation of the queuing time. In the following experiments, we adjust C (of the second part) from 0 to 4 to observe the effect of different timeout thresholds. The performance of the different strategies is shown in Fig 8, and the corresponding retry ratio ρ is shown in Fig 9. Roughly speaking, when C decreases, no matter which strategy is used, the performance is improved while the retry ratio increases.

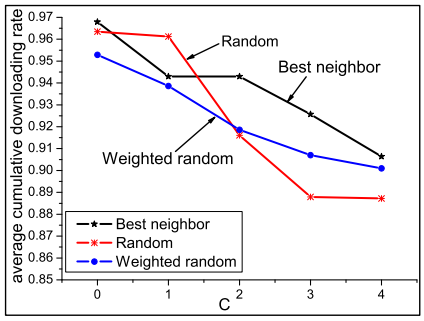

Fig. 8. The impact of different thresholds on the performance

B. Heterogeneous peers

In practice, the uplink capacity of peers can vary quite a lot, between ADSL versus Ethernet based access technologies. We expect this to affect the Random strategy,

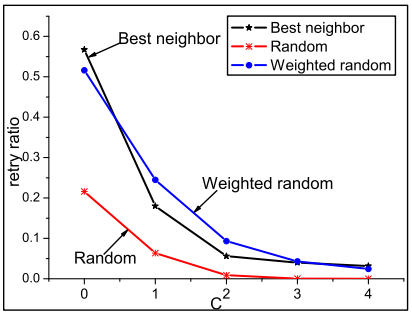

Fig. 9. The impact of different thresholds on the retry ratio

and that is the raison d'etre for the Weighted Random. We experimented with two heterogeneous networks, each with three type of peers with equal proportion. In the first network, the three types of peers had uplink bandwidths of $(0.5, 1, 1.5)$ relative to the playback rate; in the second network, the uplink bandwidths are $(0.2, 0.4, 2.4)$. In both cases, the average uplink capacity still equal to the playback rate, as in the homogeneous network case. As shown in Fig 10, the Random strategy has poorer performance in the heterogeneous case compared to the homogeneous case, as expect. The Weighted Random strategy, however, fares quite well in both cases, as is evident from Fig 11. From Fig 12, we can see that the heterogeneous networks also cause trouble for the Best Neighbor strategy. This is because the peers with small uplink bandwidth are more likely to be abandoned by requesting peers.

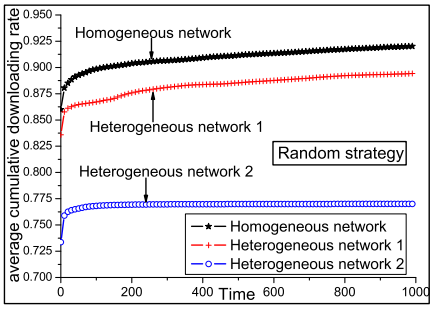

Fig. 10. The performance of the random strategy in different networks

C. Request window size

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

1) increasing the load
2) 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 keeping the load constant is at

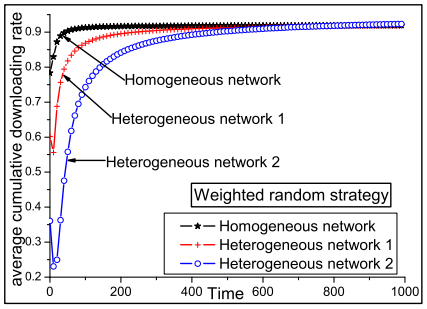

Fig. 11. The performance of the weighted random strategy in different networks

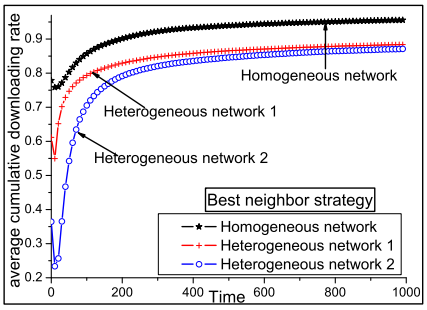

Fig. 12. The performance of the best neighbor strategy in different networks

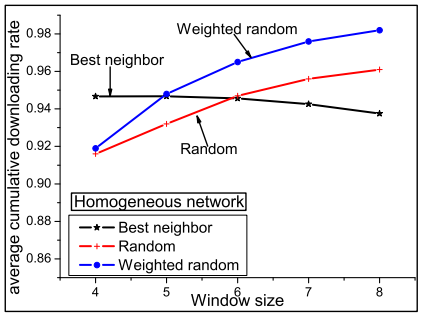

Fig. 13. The impact of window size to the performance in homogeneous network

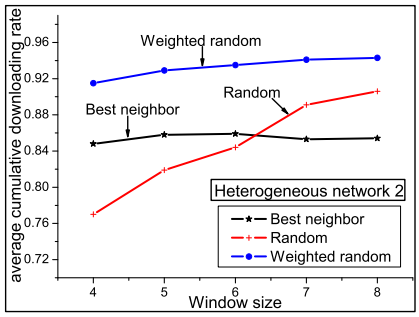

Fig. 14. The impact of window size to the performance in heterogeneous network

the expense of some additional overheads. The second effect (of increasing W) has clear implication for load balancing.

We experimented with different values of request window size for both homogeneous and heterogeneous networks. In the homogeneous case, as we increase W from 4 to 8, there is noticeable improvement for both Random and Weighted Random, as shown in Fig 13. For Best Neighbor however, there is not a significant effect. In the heterogeneous case where Random normally has a problem, the effect of a larger request window is dramatic: it lets Random achieve about the same performance as Weighted Random, as shown in Fig 14 for the second heterogeneous network. 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 more balanced load.

In Fig 15, we show the performance of the three strategies, setting the request window size to maximum. This time, we consider both the cumulative downloading rate as well as the convergence time, which can be an important metric in more dynamic networks. We can see that the simple Random strategy with large W is quite competitive.

D. Performance with adequate capacity

In all the experiments so far, we considered only cases when the average uplink capacity is equal to 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 uplink capacity (of each peer) to 1.2; in the heterogeneous case, we set the uplink capacity of the three types of peers to be $(0.5, 1, 2.1)$ respectively (with average of 1.2 as well). For performance metrics, we consider both the average cumulative downloading rate (vertical axis), as well as the convergence speed (horizontal axis). So the perfect performance is the point $(0, 1)$. The result is shown in Fig 16. Overall, the Weighted Random seems more stable all-rounded. The Random strategy does not fare its best because in this comparison $W = 4$ is used. The Best Neighbor is a little disappointing in the heterogeneous case. It does not achieve the playback rate because the peers with low bandwidth (0.5) tend to be abandoned.

VI. DISCUSSIONS AND CONCLUSIONS

A. Related Works

The load balancing problem studied in this paper arises in unstructured P2P content distribution systems, which is very popular in recent years due to success deployment of many large-scale systems such as BitTorrent [1], CoolStreaming[2], PPLive[3],SopCast[4],and TVants[5]. There is already a body of literature in modeling and analysis of such systems. On the

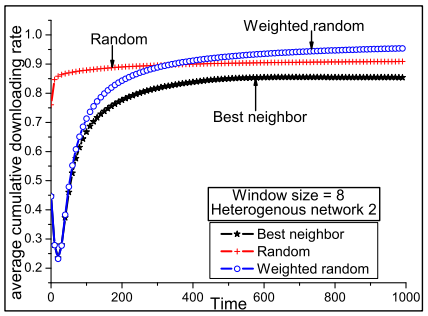

Fig. 15. The comparison between strategies given W=8 in heterogeneous network

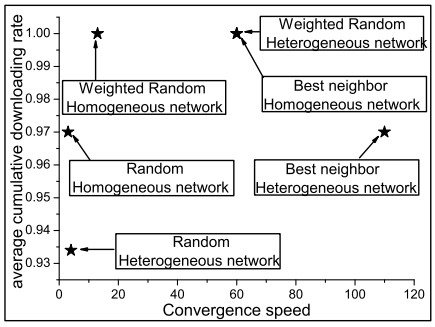

Fig. 16. Comparison of different strategies in the network with enough uplink capacity

one hand, a number of papers are devoted to understanding the theoretical capacity and other properties of such systems, such as [6], [9],[10], [14] and [11]. On the other hand, other papers try to model the algorithms in realizing such P2P systems. There are two important types of algorithms for unstructured P2P systems. The first type ensures peers have content so that they can help each other, which is often referred to as piece selection algorithms. For example, [13] and [12] are examples of modeling such algorithms and mechanisms. The other type of algorithms deal with load balancing, and are often referred to as request scheduling or load balancing algorithms, which is the subject of this paper. Finally, various paper also give a general description of these algorithms, for example, [1] for p2p file downloading systems; [2] for p2p streaming systems; and [16] for p2p Video-on-Demand systems.

Load balancing is not a new topic, and it has been studied in many contexts. For example, an elegant analysis is found in [17] for studying algorithms for balancing the load on different servers in client server systems based on the ball-and-bin model. In a p2p system, the load balancing problem is more distributed in nature and deserves new attention.

B. Conclusions and future directions

We feel that the load balancing algorithm is a generic and important component of unstructured p2p content distribution systems. In this paper, we have presented a preliminary study of some basic strategies for approaching this problem. Primarily through simulations, we are able to derive important insights. For example, knowledge about load may not always lead to better performance and robustness; and randomization can often lead to simple and effective algorithms. Our exploration considers many practical mechanisms, such as how to use timeout and try, and window of outstanding requests to improve the algorithms.

For future studies, we would like to derive analytical results for some of the conclusions. Also, we want to look into the dynamic population case and load balancing for file downloading.

VII. ACKNOWLEDGMENTS

The authors would like to thank Trevor Ng, and Yan Huang for discussions on how engineers think about when implementing algorithms for load balancing in P2P systems.

REFERENCES

[1] BitTorrent protocol. http://www.bittorrent.com/protocol.html L.
[2] 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*, 2005.
[3] "PPLive", http://www.pplive.com/.
[4] "SopCast", http://www.sopcast.com/.
[5] "TVants", http://www.tvants.com/.
[6] Massoulie and M. Vojnovic. "Coupon replication systems". In *Proc. ACM SIGMETRICS*, 2005.
[7] J. Mundinger, R.R. Weber and G. Weiss, "Analysis of Peer-to-Peer File Dissemination amongst Users of Different Upload Capacities". In *Performance Evaluation Review*, Performance 2005 Issue.
[8] R. Kumar, Y. Liu, and K. Ross. "Stochastic Fluid Theory for P2P Streaming Systems". In *Proc. INFOCOM* 2007.
[9] D. Qiu and R. Srikant. "Modeling and performance analysis of bittorrentlike peer-to-peer networks". In *Proc. ACM SIGCOMM*, 2004.
[10] B. Fan, D.M. Chiu, J.C.S. Lui, "The delicate tradeoffs in bittorrent-like file sharing protocol design". In *Proc. ICNP*, 2006.
[11] M. Lin, et al., "Stochastic analysis of file-swarming systems". In *Performance Evaluation* (2007),
[12] Y. Zhou, D.M. Chiu, J.C.S. Lui, "A simple model for analyzing P2P streming protocols". In *Proc. ICNP*, 2007.
[13] S. Sanghavi, B. Hajek, and L. Massoulie, "Gossiping with multiple messages". In *IEEE Transactions on Information Theory*, vol.53, December 2007, pp. 4640-4654.
[14] S. Liu, R. Zhang-Shen, W. Jiang, J. Rexford, M. Chiang, "Performance Bounds for Peer-Assisted Live Streaming". In *Proc. ACM SIGMETRICS*,2008.
[15] J. Li, Philip A. Chou and C. Zhang "Mutualcast: An Efficient Mechanism for Content Distribution in a Peer-to-Peer (P2P) Network ". *MSR-TR-2004-100*, Sept. 2004.
[16] Y. Huang, T.Z.J. Fu, D.M. Chiu, J.C.S. Lui and C. Huang, "Challenges, Design and Analysis of a Large-scale P2P VoD System". In *Proc. ACM Sigcomm* 2008.
[17] M. Mitzenmacher, "The Power of Two Choices in Randomized Load Balancing". In *IEEE Transactions on Parallel and Distributed Systems*, vol.12, no.10, pp. 1094-1104, Oct., 2001.
[18] B. Wang, J. Kurose, D. Towsley, W. Wei, "Multipath Overlay Data Transfer". *UMass CMPSCI Technical Report 05-45*.