Improving BitTorrent

Dimitri do B. DeFigueiredo and S. Felix Wu

August 10, 2006

Abstract

We propose two new greedy algorithms to be used by software clients running the BitTorrent peer to peer protocol: a new *peer selection* algorithm and a new *piece selection* algorithm. We provide experimental results showing that the proposed algorithms are better than the ones currently in use. When used together in our experiments, the algorithms decreased the expected download time by ??% on average when compared to an identical client running the standard *Rarest First* and *Choke/Random Unchoke* algorithms.

1 Introduction

On of the most important characteristic of peer-to-peer networks is that they can provide on-demand resource allocation. Thus, minimizing the cost of the infrastructure required to do online content distribution and enabling more people to become content publishers.

The BitTorrent peer-to-peer protocol has established itself as one of the most popular peer-to-peer file sharing systems on the Internet today[cite]. This popularity has spurred a surge of activity characterizing the system's potential for online content distribution [4, 1, 5]. In this work, we use the findings obtained to design an improved BitTorrent client.

The behavior of a peer running the BitTorrent peer-to-peer protocol is mostly determined by two algorithms: a peer selection algorithm and a piece selection algorithm. The *peer selection* algorithm determines who the local peer will cooperate with; *i.e.,* who it will download from and upload to. The *piece selection* algorithm determines which pieces of the file being downloaded should be obtained from which peer as the download progresses.

The current version (version 4) of the BitTorrent "mainline" client and most others use *Choke/Random Unchoke* as their peer selection algorithm and *Rarest First* as their piece selection algorithm[source code]. **Go on to describe the algorithms...**

Set out paper outline.

2 The peer selection algorithm

One the reasons for the success of BitTorrent is its incentive structure that penalizes free riding and promotes cooperation. Central to this, is each peer's willingness to upload. Peers are interested in downloading but may not have the intent to upload, degrading overall system performance. In this scenario, we see each peers upload bandwidth as a scarce resource. In fact, we assume that each peer wants to maximize its download bandwidth given its own upload bandwidth.

The local peer needs to decide what upload rates to give to each of the P other peers it is connected to. The current BitTorrent implementation will typically be connected to 40-80 peers [4], but only upload to 4 of them at any particular time. It is claimed that uploading simultaneously to a larger number of peers will have a detrimental effect on TCP congestion control and decrease the total uploading rate [2]. We are unaware of any results that substantiate such claim; therefore, we placed no such restrictions on the number of peers our client can connect to. In fact, we go one step further, *adaptively* throttling each individual peer connection to different uploading rates. The current implementation of the mainline BitTorrent client only establishes an overall bound for all peers and does not throttle peers individually. Other clients, establish fixed upper bounds per peer but not adapt.

2.1 The Model

We assume that the local client is playing a game in a series of rounds, each round lasting for about 10 seconds. The local client's job is to download the file as quickly as possible. In each round, we have available a total upload bandwidth of U bps and we give each peer a u_i fraction of this bandwidth. Clearly,

$$\sum_{i=1}^{P} u_i \leq 1$$

We use an inequality (instead of an equality) here to guarantee that we can only increase our gains by having more bandwidth available, as we can always fall back to a solution that does not use all the available bandwidth. Let us also assume that the local peer is given a set of P (groups of) probability density functions $p_i(d_i, u_i)$, one for each peer. For a given u, the function $p_i(d, u)$ represents the probability that peer i will reciprocate an upload rate of u with the download rate d. Furthermore, we assume that each p_i stems from independent events, so that the download rate received from peer i is only dependent on peer i and the upload rate u_i given to it, but **not** on the upload rate given to any other peer.

If we assume that each round is independent of the next then getting the maximum outcome in one round will not affect how much we get in the next. Moreover, our upload rate is a perishable good, *i.e.* we cannot save upload rate for later use. Given these two assumptions, we can use a greedy strategy

to obtain the optimal overall strategy over many rounds. We simply need to maximize the pay-off function for each round.

The payoff function for a single round is simply the sum of the download rates d_i obtained from each peer i in that round:

$$O = \sum_{i=1}^{P} d_i$$

Thus, the expected return in a round is given by:

$$E[O] = E\left[\sum_{i=1}^{P} d_i\right] = \sum_{i=1}^{P} E[d_i]$$

and we need to maximize this expression subject to the constraint on u_i given above and $0 \leq u_i \leq 1$. Notice that under this model each peer is completely characterized by the download rates it is expected to provide for each u_i. In other words, peers are described by the function $d_i(u_i) = E[d_i|u_i]$ which provides the expected download rate provided as a function of the upload rate received. Because of the inequality used in our constraint, the expected pay-off is a non-decreasing function of the *overall* available upload rate.

The above maximization problem can be solved for many different forms of $E[d_i|u_i]$. We will look at the following special cases that provide insight into the solutions:

- $E[d_i|u_i]$ are step functions in u_i.
- $E[d_i|u_i]$ are linear in u_i **to do (boring) insight into ideal number of peers**.
- $E[d_i|u_i]$ are concave non-decreasing in u_i.**still to do**
- $E[d_i|u_i]$ are convex non-decreasing in u_i.**still to do**
- We know nothing about the probabilities $p_i(d_i, u_i)$ and therefore $E[d_i|u_i]$has the following form...**I wish!**

A non-decreasing expectation in u_i means that it is not possible to get a lower download rate by uploading more.

2.2 A Selfish Upload Ratio Algorithm

Current behavior of bittorrent clients can be modeled by considering the expected download rate function $d_i(u_i) = E[d_i|u_i]$ as a step function of amplitude $\tilde{d}_i$ with the step centered at $\tilde{u}_i$. The standard implementation of the peer selection algorithm ranks each remote peer according to the download rate they have provided in the last round. The local peer continues its upload to the three fastest peers. More precisely, to the three peers that provided it with the highest download rates. Unless it has just recently (been unchoked and) started

to receive pieces from the local peer, the fourth fastest peer will stop receiving any available bandwidth and a new randomly selected peer will be unchoked and start to download from the local peer.

Clearly, if the local peer is always one of the three fastest uploaders to each of its remote peers it will continue to receive service from them. The local peer can estimate the overall download rate received from each of its remote peers by timing the *have* messages that are received after the remote peer completes the download of a piece. Dividing this value by 3 we obtain an upper bound on the download rate provided by the third fastest peer of that remote client. We will use this upper bound as an estimate of $\tilde{u}_i$. We propose that the local peer should rank its remote peers at each round and greedily upload to the peers that provide it with the best $\tilde{d}_i/\tilde{u}_i$ ratios, throttling each connection individually. In this way, the local client can minimize how much upload bandwidth it gives to each of its remote peers and at the same time maximize how much download rate it gets. One natural consequence of using this algorithm is that the local peer will always ask for pieces from seeds (to which it does not upload).

In this model, choosing who to upload to at each round is equivalent to solving the knapsack problem. The total upload bandwidth available is the maximum allowable weight. The value of each item is the download rate we are getting from the corresponding peer and the weight of each item is the corresponding upload rate we should provide. Note that the greedy strategy of simply picking the peers with the highest download to upload ratio is a 2-approximation to this problem [cite] (and optimal for the fractional knapsack).

There is a lot more to write, but this has always been the main idea. There maybe a better strategy using restless bandits with switching costs, I need to look more into that.

3 The piece selection algorithm

To do: Show that luck by itself is a very good code generator

Results in [4] suggest that the *Rarest First* piece selection algorithm is sufficient for the good operation of BitTorrent as it ensures that each peer is always interested in *all* other peer. Or as [4] put it: "We say that there is ideal entropy when each leecher is always interested in any other leecher.". This is desirable because it implies that the upload rate and not piece availability is the main criterion used by peers to choose who they cooperate with. The results in [4] found that a client using *Rarest First,* was almost all the time interested in over 80% of all other peers in 75% of the torrents studied. Despite these results we feel that there is room for improvement in the piece selection algorithm for three reasons:

1. *Rarest First* was found to be very effective when used in combination with BitTorrent's current peer selection strategies. In particular, the current random unchoke strategy may provide for a very good randomization of the available pieces by itself.

2. In 25% of the torrents studied *Rarest First* was not effective in keeping the local peer interested in all other peers and Legout *et al.* identified a *first blocks problem* that our previous research suggests can be aggravated if peers are using a non-exploitable bitwise tit-for-tat strategy[6]. In such cases, it is desirable to channel new pieces available from the source through the highest capacity peers as quickly as possible. Therefore, some non-random bias in piece selection may be desirable.

3. In the other 75% (in fact for this there was a smaller fraction of 70%) of the torrents studied by Legout *et al.*, it is not clear whether the 80% of the peers that were interested in the local client were precisely the ones that could give it the highest upload rate. It is not clear whether each client is able to maintain the peers that provide it with the highest upload rate always interested, or whether such peers were precisely the ones that became uninterested.

We restrict our client so that it downloads each piece from only a single peer. This enables us to assign blame and avoid being tricked by clients that do not actually have the pieces they advertise. The piece selection algorithm is crucial at the beginning of the download when a peer needs to get pieces that enable it to upload to others. We now describe our model of the piece selection problem and a new piece selection algorithm. **TODO: Optimistic simultaneous download exception.**

3.1 The MDP model

We model the piece selection problem as a **M**arkov **D**ecision **P**rocess (MDP). Assume that the local peer downloads pieces at a constant rate (*e.g.,* 1 piece every 5 seconds) and normalize the download rate received to 1, so that at each new unit of time it gets a new piece. Pretend to be the local peer for a minute. For each peer i who is currently uploading to us, we keep track of *the pieces we have that the remote peer i does not have.* These pieces form a queue Q_i. We add a piece to this queue every time we download a piece that peer i does not have. Thus, we can add a maximum of one piece per unit time to each queue. Remote peers remove pieces from their respective queues every time they obtain a piece that they did not have but we did. We pay a penalty any time a queue is empty. More precisely, we pay a penalty *per unit time* for each queue that is empty and each queue Q_i has its own penalty rate C_i. This penalty rate is proportional to the download rate currently obtained from that peer. We discount future penalties at a rate $\delta < 1$. We neglect that the local peer may establish connections to different peers throughout its download and fix the total number of peers that the local client main connect with to P. The length of each queue, *i.e.* the number of elements in each queue Q_i at each time t, is denoted by $l_i(t)$ or simply l_i.

The "real world" problem of deciding which piece to download from other peers is modeled as choosing in which queue to place the piece obtained in the

last unit of time[1]. For now, we will assume that each piece we obtain can be put in any of the queues. Thus, in our model, at each new timestep the local peer has to decide on which queue Q_i to put the piece obtained in the last unit of time and in so doing the local peer wishes to minimize the chance that any of the queues will become empty. In fact, it wants to minimize the expected overall cost:

$$\sum_{i=1}^{P} E[\text{cost of } Q_i]$$

where the expected cost of queue Q_i is given by:

$$E[\text{cost of } Q_i] = \int_0^\infty C_i \delta^t Prob(Q_i \text{ is empty at time } = t)\, dt$$

Clearly, the probability that Q_i is empty at time t depends on when we put extra pieces in this queue. Therefore, the local peer needs to choose a policy to minimize its overall incurred costs. This policy is precisely the piece selection algorithm we wish to determine. The algorithm proposed here is a simple extension of the policy proposed by Whittle to address the *restless bandits* problem [7]; which, in turn, was an extension of the policy proposed by Gittins for the multiarmed bandit problem [3]. Gittins showed that this policy is optimal for the multiarmed bandit problem [3]. Gittins' policy is very simple. After calculating an index for each project, each queue in our case, we make a greedy choice and simply pick the queue with the highest index at each time interval. The nature of the Gittins index is such that it incorporates in its value the trade-off in future gains and makes the greedy strategy based on it optimal.

For our particular problem, the Gittins index for queue Q_i is given by the difference in expected cost of Q_i under two circumstances:

- When *only one* piece is immediately added to the queue.
- When no more pieces are added to the queue.

It should be clear that this policy maximizes at each step the expected gain that can be obtained by adding a piece to one of the queues. Unfortunately, we cannot show that this strategy is optimal for the *restless bandit* problem. But it is asymptotically optimal, see [7].

Let us first calculate the expected cost of Q_i assuming that:

- Q_i has initial length $l_i(0)$,
- pieces are removed according to a Poisson points process of rate λ_i; and,
- no more pieces are added to the queue.

[1] This introduces some delay issues that we will address later.

Under these assumptions, the probability that Q_i is empty at time t is the same as the probability that at least $l_i(0)$ pieces are removed from Q_i by time t.

$$E[\text{cost of } Q_i] = \int_0^\infty C_i \delta^t Prob(\text{at least } l_i(0) \text{ pieces are removed from } Q_i \text{ by time } t)\, dt$$

$$= \int_0^\infty C_i \delta^t [1 - Prob(\text{ less than } l_i \text{ pieces are removed by time } t)]\, dt$$

$$= \int_0^\infty C_i \delta^t \left[1 - \sum_{j=0}^{l_i - 1} Prob(\, j \text{ pieces are removed by time } t) \right] dt$$

$$= \int_0^\infty C_i \delta^t \, dt - \int_0^\infty C_i \delta^t \sum_{j=0}^{l_i - 1} e^{-\lambda_i t} \frac{(\lambda_i t)^j}{j!}\, dt$$

The last equality follows from the properties of a Poisson points process of rate λ, for which the probability that there are exactly k points in a time interval of length t_a is given by $e^{-\lambda t_a} \frac{(\lambda t_a)^k}{k!}$. Evaluating the first integral and changing the order of the finite sum with the second integral we obtain:

$$E[\text{cost of } Q_i] = C_i \left[\frac{e^{t \ln \delta}}{\ln \delta} \right]_0^\infty - C_i \sum_{j=0}^{l_i - 1} \frac{\lambda_i^j}{j!} \int_0^\infty \delta^t e^{-\lambda_i t} t^j \, dt$$

We have that $\delta < 1$ so $\ln \delta$ is negative and we can write:

$$E[\text{cost of } Q_i] = \left[-\frac{C_i}{\ln \delta} \right] - C_i \sum_{j=0}^{l_i - 1} \frac{\lambda_i^j}{j!} \int_0^\infty e^{-(\lambda_i - \ln \delta) t} \, t^j \, dt$$

Making a change of variables in the integral for $\tau = (\lambda_i - \ln \delta) t$ we obtain:

$$E[\text{cost of } Q_i] = \left[-\frac{C_i}{\ln \delta} \right] - C_i \sum_{j=0}^{l_i - 1} \frac{\lambda_i^j}{j!} \left(\frac{1}{(\lambda_i - \ln \delta)^{j+1}} \int_0^\infty e^{-\tau} \, \tau^j \, d\tau \right)$$

Now, recall that for $j \in \mathbb{N}$:

$$\int_0^\infty e^{-\tau} \, \tau^j \, d\tau = \Gamma(j+1) = j!$$

Therefore, we have.

$$E[\text{cost of } Q_i] = -\frac{C_i}{\ln \delta} - C_i \sum_{j=0}^{l_i - 1} \frac{\lambda_i^j}{j!} \left(\frac{j!}{(\lambda_i - \ln \delta)^{j+1}} \right)$$

$$= -\frac{C_i}{\ln \delta} - \frac{C_i}{(\lambda_i - \ln \delta)} \sum_{j=0}^{l_i - 1} \left(\frac{\lambda_i}{\lambda_i - \ln \delta} \right)^j = -\frac{C_i}{\ln \delta} - \frac{C_i}{(\lambda_i - \ln \delta)} \frac{\left(\frac{\lambda_i}{\lambda_i - \ln \delta} \right)^{l_i} - 1}{\left(\frac{\lambda_i}{\lambda_i - \ln \delta} \right) - 1}$$

Finally, simplifying the expression above we obtain:

$$E[\text{cost of } Q_i] = -\frac{C_i}{\ln \delta} \left(\frac{\lambda_i}{\lambda_i - \ln \delta} \right)^{l_i}$$

The Gittins index for Q_i can then be calculated as the difference in expected cost for different queue lengths (here we omit the i indices):

$$\nu(i) = E[\text{cost of } Q \,|\, \text{length is } l] - E[\text{cost of } Q \,|\, \text{length is } l{+}1] = \frac{C}{\lambda} \left(\frac{\lambda}{\lambda - \ln \delta} \right)^{l+1}$$

Notice that all parameters, including the discount factor δ, are specific to the queue in question. In fact, it may be desirable to have a different discount factor for each queue (*i.e.* peer) depending on the peer's reliability.

3.2 Applying the MDP solution

To obtain the above strategy of greedly putting a piece in the queue Q_i that has the highest index we used a somewhat simplified model of the piece selection problem. We now consider how we adapt the strategy obtained to address all the nuances of the piece selection problem.

In the simplified model used above, we assumed that each piece could only be put in one queue at the time. This models the real world situation in which a downloaded piece is only of interest to one other peer at a time. This situation is more likely the exception rather than the rule in a real-world setting. To change this to a more general setting where a piece can satisfying many peers simultaneously, we assign indexes to pieces rather than to queues. This is done simply by summing the index of the queues to which a piece will be added once downloaded. For example, if peers 1, 2 and 5 do not have piece A. Then, the index of piece A will be given by: $\nu(A) = \nu(1) + \nu(2) + \nu(5)$. In this way, we can establish an ordering of pieces that tells us which pieces are more desirable. Again, we choose the piece with the highest index.

We made another two important simplifications in our model. We assumed that each piece obtained could be put in any queue. In some sense, that is analogous to assuming that peers can provide us with any piece we desire. In reality, each time we have to decide which piece to download we can only pick from a subset of all the pieces available. This is the subset of pieces that we do not have that the peer we are downloading from has. Thus, sometimes the most desirable piece may not be available. We will simply use a greedy strategy and pick the most desirable piece that is available to us at any instant in time. It is easy to construct counter examples that show that this strategy is not optimal;

however, because of the dependencies between successive pieces it is not clear how to go about finding such optimal strategy. In the case when the queue lengths do not change or change very little after each piece is selected we can use a bipartite matching algorithm between pieces and available download spots to find the optimal download schedule.

The other simplifying assumption is that pieces are obtained at a constant rate. In fact, we download pieces from different peers each of which provide us with a different upload rate. We have to decide which piece to download from each peer a few seconds before the piece is actually obtained. The time it takes for a piece to be downloaded depends on the upload rate the remote peer is willing to provide us. The key idea that enables us to circumvent this problem is that we can extrapolate the current download rate obtained from each remote peer and estimate the finishing time for each piece we are currently downloading. Thus, we are able to ask to download the most desirable pieces according to their download finishing time. For example, assume that the two most desirable pieces are currently A and B respectively. We can download them from peers 1 (fast) or 2 (slow) which provide us with the rates of 1 piece every 5 seconds and 1 piece every 10 seconds respectively. Peer 1 is currently busy uploading another piece and will be done in 3 seconds but we have to decide which piece to download from peer 2 now. If we delay downloading piece A until peer 1 is ready we will be able to obtain the piece in 8 seconds. On the other hand, if we ask for it now it will only be completely downloaded after 10 seconds, whereas piece B will be done in 8 seconds. Clearly, we are assuming that the download rates do not change significantly; but, that being the case, it should be clear that we want to select pieces such that the most desirable piece obtains the earliest download finishing time.

One final modification: we should have criterion 3(keep others that i am interested in (but who do not yet upload) interested in me.**) kick in when the queues are too long.**

Notice that the above piece selection algorithm takes into account the download rates obtained from peers and decides not only which piece to download, but also who to download it from. Also, if all queues are identical and pieces are obtained from all peers with the same delay then the algorithm reduces to an implementation of *Rarest First* where only peers that upload to the local peer "get to vote".

4 Experimental Results

describe the side-by-side testing we wish to do here:

- write the client
- deploy two clients side by side on the same torrents
- repeat experiments to average out random fluctuations.

5 Conclusion

concluding remarks.

References

[1] A. R. Bharambe, C. Herley and V. N. Padmanabhan, "Analyzing and Improving BitTorrent Performance", Technical Report MSR-TR-2005-03, Microsoft Research, USA, Feb 2005him if.

[2] BitTorrent Protocol Specifications. http://bittorrent.org/protocol.html (as of 10-Jul-2006)

[3] J. C. Gittins, *Multi-armed Bandit Allocation Indices*, John Wiley, 1989.

[4] A. Legout, G. Urvoy-Keller and P. Michiardi, "Rarest First and Choke Algorithms Are Enough", *Technical Report* (inria-00001111, version 2 - 1 June 2006), INRIA, Sophia Antipolis, June 2006.

[5] N. Liogkas, R. Nelson, E. Kohler and L. Zhang, "Exploiting BitTorrent For Fun (But Not Profit)", *5th International Workshop on Peer-to-Peer Systems, IPTPS*, Santa Barbara, USA, 2006.

[6] Our own p2p-"framework" paper (not this one).

[7] P. Whittle, "Restless Bandits: Activity Allocation in a Changing World", In J. Gani, editor, vol. **25A** of *Journal of Applied Probability*, pp. 287-298. Applied Probability Trust, 1988.