Improving Robustness of Peer-to-Peer Streaming with Incentives

Vinay Pai Alexander E. Mohr

Stony Brook University
{vinay, amohr}@cs.stonybrook.edu

Abstract

In this paper we argue that a robust incentive mechanism is important in a real-world peer-to-peer streaming system to ensure that nodes contribute as much upload bandwidth as they can. We show that simple tit-for-tat mechanisms which work well in file-sharing systems like BitTorrent do not perform well given the additional delay and bandwidth constraints imposed by live streaming. We present preliminary experimental results for an incentive mechanism based on the Iterated Prisoner's Dilemma problem that allows all nodes to download with low packet loss when there is sufficient capacity in the system, but when the system is resource-starved, nodes that contribute upload bandwidth receive better service than those that do not. Moreover, our algorithm does not require nodes to rely on any information other than direct observations of its neighbors' behavior towards it.

1 Introduction

In recent years, BitTorrent [4], a peer-to-peer file sharing protocol has become one of the most widely used tools for bulk data dissemination to large numbers of nodes. BitTorrent allows a large number of nodes to simultaneously download a large file by breaking it into chunks and having different nodes exchange chunks among each other.

While BitTorrent is effective for transferring files, the file is not downloaded in sequence and is therefore generally unusable till the download is complete. Moreover the bandwidth delivered often varies over time, making it unsuitable for applications like streaming video.

Many peer-to-peer streaming protocols [2, 6, 5, 10] have been proposed. However, in order to perform well, most of them assume a resource-rich environment where there is sufficient upload capacity in the system to support all downloaders.

However, several studies [1, 11] have shown that users of peer-to-peer networks tend to be selfish and try to benefit from a system without contributing resources in return. Moreover, several nodes are *unable* to contribute as much upload bandwidth as the download bandwidth they consume because they are using an asymmetric Internet connection like a consumer cable-modem or ADSL line.

This could result in a system where the demand for download bandwidth exceeds the supply of available upload bandwidth, making it impossible to satisfy all demand. In such a system we would like to give nodes an incentive to upload as much as they can by making the probability of suffering packet loss inversely proportional to the upload bandwidth contributed.

To achieve this, we tried to design an incentive scheme for peer-to-peer streaming on top of the Chainsaw [9] streaming protocol. We found that due to the strict time and bandwidth constraints and intolerance to long delays in streaming, incentive schemes designed for file-sharing protocols perform very poorly in streaming systems. In this paper, we start by presenting some of our *unsuccessful* attempts along with our reasoning of why they failed to perform well, in the hope of generating discussion.

In addition, we present an incentive scheme called Token Stealing based on an Iterated Prisoner's Dilemma that appears promising based on our preliminary experiments on PlanetLab [3].

2 Background

We implement our incentive schemes on top of Chainsaw [9], a streaming protocol based on an unstructured mesh network. Chainsaw uses a simple request-response protocol which is briefly described here.

Chainsaw Streaming Protocol

Chainsaw is designed to deliver a stream of data from one node (called the *seed*) to a large number of recipients. While Chainsaw may be generalized to multiple seeds, as well as many-to-many multicast, in this paper we only consider one-to-many multicast with a single seed. We refer to the set of nodes to which a peer is connected as its *neighbors*.

Every node maintains a list of packets that its neighbors are willing to provide. To ensure that this list is updated properly, whenever a node receives a new packet, it broadcasts a NOTIFY message to its neighbors to inform them of the change. The seed obviously does not receive packets, but does generate new packets periodically. The seed sends out NOTIFY messages to its neighbors every time it generates a new packet.

Every node also maintains a *window of interest*, which is the set of sequence numbers that the node is interested in acquiring at the current time. The node slides its window of interest forward over time as new packets stream in. If a packet has not been received by the time it "falls off" the trailing edge of the window, the node will consider that packet lost and will no longer try to acquire it.

For every neighbor, a node creates a list of *desired packets*, i.e. a list of packets that the node requires that the neighbor is able to provide. It will then pick one or more packets from the list at random and request them via a REQUEST message.

A node keeps track of which packets it has requested from which neighbor to ensures that it does not request the same packet from multiple neighbors. It also limits the number of outstanding requests with a given neighbor to ensure that requests are spread out over all neighbors. Finally, when a node receives a REQUEST message, it responds with a corresponding DATA packet as bandwidth permits.

Chainsaw has no global routing tables, so it does not depend on any specific network topology. In this paper, we will assume a topology in which every node repeatedly connects to a randomly picked node from the list of known hosts until a predefined minimum number of neighbors (the *node degree*) is reached. This network has the advantage of being very easy to construct and maintain even in the face of the sudden departure of a large fraction of nodes. Many practical peer-to-peer networks like BitTorrent and Gnutella use an unstructured random graph topology.

3 Poor Performance: Naïve Tit-for-Tat

Chainsaw's request-response protocol is similar to that used by BitTorrent, as well as SWIFT[12], a pairwise currency mechanism for file-sharing that we showed to be more effective at providing fairness in file-sharing than BitTorrent's incentive mechanism.

In SWIFT, every node maintains *credit* for each of its neighbors and honors packets requests only when the neighbor has enough credit. Whenever it receives a packet from a neighbor, the node extends it α packets worth of credit. In addition, trading is jump-started by initializing neighbors with γ packets worth of credit instead of zero, and deadlocks are avoided by periodically extending nodes a small fraction β of their total upload capacity in credit every second, regardless of data received from it.

As long as nodes consistently upload, they will keep earning credit with their neighbors and be able to download. However, nodes that do not upload will soon deplete their credit with their neighbors and not be able to download anymore, except for small trickle of free credit they receive from their neighbors in the form of β.

While SWIFT was very effective at ensuring fairness in file-transfer applications, we found a similar mechanism to perform very poorly when applied to streaming. In our simulations we found that over time, a large fraction of nodes started to suffer severe ($> 50\%$) packet loss even in a system where every node tried to upload as much as their capacity allowed. This was caused by small imbalances between nodes (eg. due to different delay characteristics, distance from seed, number of neighbors) being amplified by an undesired positive-feedback loop.

For instance, consider a pair of nodes A and B, where A is closer to the seed than B. In this situation, node A is likely to receive new packets before node B. As a result, node B has fewer opportunities to upload packets to node A, resulting in a net loss of credit. Eventually, node B runs out of credit and is no longer able to download from node A. However, given its proximity to the seed, it is likely that node A was a source of packets that was of interest to node B's other neighbors. Therefore, the loss of node A as a trading partner puts node B in a less favorable position to trade with the rest of its partners. This creates a positive feedback loop where a slight disadvantage is ultimately amplified to the point where a node is unable to earn enough credit to avoid packet loss.

4 Partial Success: Compensating for Trading Imbalances

We experimented with a number of mechanisms for compensating for these small imbalances.

4.1 Preferential Uploading

In the naïve tit-for-tat experiments, we found that nodes ran out of credit because they were unable to upload enough packets to some of their neighbors to maintain a stable supply of credit. Therefore, we implemented a system where nodes aggressively tried to upload to neighbors they were running out of credits with by giving requests from those neighbors a higher priority. Quickly satisfying existing requests results in more requests for packets from that neighbor because nodes limit the number of requests outstanding with a given neighbor at any time.

Unexpectedly, this strategy made the problem *worse*. Some nodes were at an advantage with respect to most of their neighbors, which led to an "arms race" among neighbors to upload as quickly as possible to the advantaged node. This created a new positive feedback loop where advantaged nodes were put at an increasingly greater advantage by neighbors aggressively uploading to them. Eventually, the neighbors that lost the arms race ran out of credits as they did in the naïve tit-for-tat system.

4.2 Advantaged Nodes Back Off

Our next approach was to have advantaged nodes attempt to *reduce* the number of packets they uploaded rather than increase the number of packets their disadvantaged neighbors uploaded to them. We did this by having nodes keep track of every neighbor's balance ratio,

$$\text{balance-ratio} = \frac{\text{total-download}}{\text{total-upload} + \text{total-download}}$$

Note that this calculation is done purely based on direct local observations of a neighbor's behavior towards the node.

Nodes with balance-ratio < 0.5 have been uploading more than they have been downloading. By default, nodes send NOTIFY messages to all their neighbors when they receive a new packet to enable them to download it. However, as the balance-ratio fell below 0.5, we linearly reduced the number of neighbors notified. This ensured that the advantaged nodes only uploaded a small number of copies of every packet.

This is beneficial to the advantaged node, disadvantaged node, and the system as a whole. The advantaged nodes benefit by having some of the burden of uploading taken off them, while the disadvantaged nodes benefit by having a greater opportunity to upload packets to their neighbors and earn credit. The overall amount of upload bandwidth in the system is generally not reduced because some of the burden of uploading packets is shifted from the advantaged to the disadvantaged nodes. The advantaged node is then able to use its upload bandwidth to rapidly propagate new packets rather than multiple copies of old packets.

This scheme worked very well in our simulations, and we were able to maintain a balance-ratio between 0.45 and 0.55 across all pairs of neighbors in the network. Unfortunately, the algorithm failed to produce a significant benefit in real-world tests on real a implementation of the protocol on the PlanetLab testbed. We found that the variation in bandwidth capacity and round-trip delays between different pairs of nodes in the network so great that it was not possible to accommodate the slowest nodes without dragging down the performance of the entire system.

5 Promising: Token Stealing Algorithm

Our next attempt was an algorithm we call *Token Stealing*, which builds on the standard *token bucket* model commonly used to allocate limited bandwidth among competing processes. The Token Stealing algorithm sets up local markets at every node where neighbors compete for the node's upload capacity. When the demand for bandwidth from the node exceeds the node's capacity, nodes that upload receive preferred service, while this constraint is relaxed when there is adequate bandwidth to fulfill all requests.

We first outline the standard token bucket algorithm.

The token bucket algorithm works by having a virtual bucket into which tokens are added periodically. Whenever a packet is transmitted, an equivalent number of tokens must be removed from the bucket—packets may only be transmitted when there are a sufficient number of tokens available in the bucket. Thus, the overall bandwidth can be controlled by controlling the rate at which tokens are added to the bucket.

The number of tokens that may accumulate in the bucket is limited to some maximum value to prevent a large number of tokens from accumulating during periods when there is low demand for bandwidth.

The basic token bucket algorithm only ensures that the overall bandwidth doesn't exceed a specified limit. We augment this with the Token Stealing algorithm to give a higher priority to nodes that have been consistently uploading than those that haven not.

The **Token Stealing algorithm** is a simple extension of the token bucket algorithm. In this algorithm, every node maintains a standard token bucket that we refer to as the *shared bucket* into which tokens are added periodically. In addition, the node maintains a separate bucket for each of its neighbors. We refer to these as *private buckets*. Whenever a node receives a packet from one of its neighbors, it removes tokens from the shared bucket and transfers them to that neighbor's private bucket. This has the effect of reserving a portion of the node's upload bandwidth to repay the neighbor for the packets it has uploaded.

To prevent neighbors from reserving large amounts of bandwidth that they never utilize (for example, because they are connected to other nodes with large upload capacities), there is a limit on the size of the private buckets. Tokens that overflow the private buckets are returned to the shared bucket.

Which Bucket First?

The question of which bucket to deduct tokens from when a neighbor requests a packet is interesting. One may choose

to first deduct tokens from the private bucket and dip into the shared bucket only if there are not enough tokens in the private bucket, or one may use up tokens from the shared bucket first.

In our experiments we found that the both strategies give the neighbors that upload (and therefore have tokens in their private buckets) an advantage, but that advantage is considerably greater in the latter case. When tokens are deducted from the private buckets first, neighbors that upload do not compete in the market for the shared tokens unless their private buckets are empty. This makes it easier for neighbors that do not upload to receive a portion of the bandwidth.

When tokens are deducted from the shared bucket first, all neighbors compete equally in the market for shared tokens before dipping into their private buckets, which act as a "reserve". This amplifies the priority given to the nodes that upload.

Therefore, the strategy we choose is to deduct tokens from the shared bucket first and only dip into the private bucket when the shared bucket is empty.

5.1 Analysis

With the Token Stealing algorithm, the total upload capacity of the node is still limited by the rate at which tokens are added to the token bucket, i.e. the upload bandwidth limit. However, unlike a simple token bucket system where all nodes have an equal opportunity to use up tokens from the bucket, the Token Stealing algorithm favors neighbors that upload.

Whenever a neighbor uploads a packet to a node, the node reserves tokens for that neighbor's use. Every packet the neighbor uploads serves to a node increases the chances that the neighbor will be able to download a packet in the future.

If all neighbors upload equally, all private buckets will have the same number of tokens in them, which gives all neighbors equal priority. However, a neighbor that does not upload will not have tokens in its private bucket and will be limited to competing with other neighbors for tokens from the shared bucket.

Whether or not the non-uploading neighbor succeeds in downloading depends on the total supply and demand at that node:

5.1.1 Node has excess upload capacity

If the node has more than enough upload capacity to fulfill the demand of all of its neighbors, the shared bucket will have tokens in it and the neighbor that does not upload will still be able to download. This ensures that a node's upload capacity is utilized as much as possible.

It is possible for a few nodes, known as *free-riders* to try to leach off the system by selectively connecting to nodes with excess capacity. This strategy will work so long as the number of free-riders is small. If a large number of nodes attempt to leach off the system, they will compete among each other for tokens from the shared token bucket. This makes the effect of free-riders self-limiting.

5.1.2 Node has limited upload capacity

If the node does not have enough capacity to satisfy all requests, most of the tokens will be moved to the private buckets of the neighbors that do upload, and the shared bucket will generally be empty. As a result, the neighbors that upload will be able to use the tokens from their private buckets to download packets, but nodes that do not upload will be forced to compete for the scarce tokens from the shared bucket.

5.2 Prisoner's Dilemma

The Token Stealing Algorithm may be modeled as an Iterated Prisoner's Dilemma problem. If all of a node's neighbors defect (refuse to upload), they all share the common pool and none of the neighbors has an advantage. However, a neighbor that chooses to upload (cooperate) can "steal" tokens away from the shared bucket. The neighbor will still compete equally for the remaining tokens in the shared bucket, but will have a private reserve for itself in addition to the tokens it receives from the shared bucket. In this case, the best strategy for the other neighbors to upload in order to move tokens to their own private buckets. Thus, whenever the upload capacity at a node is scarce, the dominant strategy for every neighbor is to upload to that node, i.e. to cooperate.

6 Experimental Evaluation

To evaluate the performance of the Token Stealing algorithm, we build an application and deployed it on 350 nodes across the globe on the PlanetLab[3] testbed. All nodes joined the network before the seed (source node) started broadcasting data, and connected to an average of 15 neighbors each. The stream was divided into 4 kilobyte packets at a rate of 25 packet/second to give a total stream rate of a 100 kilobytes/sec.

We did not constrain the download capacity of nodes in any way, but capped the upload capacity of nodes to put them in one of two classes:

1. *Fast Nodes*: Maximum upload capacity = 200 kilobytes/sec

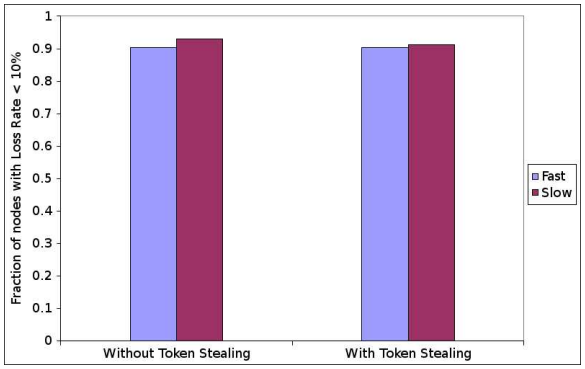

Figure 1: In a resource-rich system both fast and slow nodes are able to download with very little packet loss.

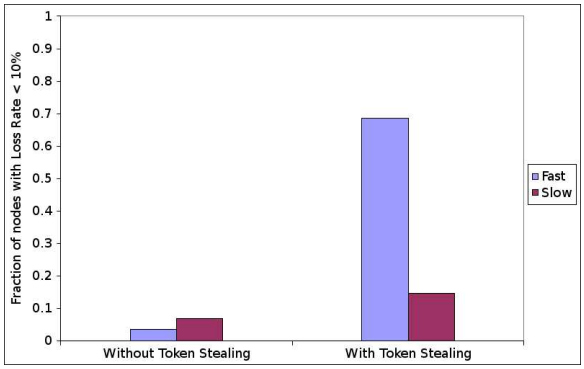

Figure 2: In a resource-starved system without Token Stealing, both fast and slow nodes suffered heavy packet loss. Barely 3% of fast nodes and 6% of slow nodes were able to download with less than 10% packet loss. With Token Stealing enabled, however, 68% of the fast nodes and 14% of the slow nodes suffered a packet loss rate under 10%.

2. *Slow Nodes*: Maximum upload capacity = 25 kilobytes/sec

6.1 Resource-Rich System

In our first experiment, the system had 75% of fast nodes and 25% of slow nodes. This made the average supply of upload capacity 156.25 kB/node. The demand from every node regardless of their upload capacity was the full 100 kB/sec stream rate. Thus, the system had approximately one and a half time the supply as demand.

We ran this system for 300 seconds and measured the packet loss rate experienced by different nodes. Note that machines on PlanetLab are shared between many researchers and are often very heavily loaded, resulting in severe and unpredictable constraints on available bandwidth and CPU time. This causes some nodes to suffer severe packet loss even when there are adequate resources in the system.

In this system, over 90% of the fast nodes as well as slow nodes suffered less than 10% packet loss regardless of whether or not the Token Stealing algorithm was used. This shows that the Token Stealing algorithm does not harm the performance of a resource-rich system.

6.2 Resource-Starved System

In our second experiment, the system had 25% of fast nodes and 75% of slow nodes. This made the average supply of upload capacity 68.75 kB/node. Thus, the system had approximately two third the supply as demand.

In this system, there is a major benefit to having the Token Stealing algorithm enabled. Without token stealing, barely 3% of fast and 6% of slow nodes had less than 10% packet loss. With Token Stealing enabled, things improved dramatically for the fast nodes—68% of them had less than 10% packet loss. Clearly, in a resource-starved system with Token Stealing enabled, nodes have a big incentive to upload as much as they can.

7 Discussion

Our investigations have shown that despite the many similarities between BitTorrent-like file-sharing systems and mesh-based peer-to-peer streaming systems, incentive schemes used in file-sharing can not be easily applied to streaming.

Simple tit-for-tat schemes do not work well because of the additional constraints imposed by live streaming. For example, in a file-sharing network, every packet is useful until all nodes in the system have downloaded that packet. In streaming, however, packets quickly become obsolete.

Our preliminary work with the Token Stealing algorithm has shown promising results. We find that it allows nodes that are unable to contribute much upload bandwidth to still download the stream with low packet loss so long as the supply of bandwidth in the system exceeds the demand. This allows the system to take advantage of altruistic nodes that contribute more upload bandwidth than the stream rate, and to avoid imposing harsh penalties on nodes that are unable to upload (for example ADSL nodes).

However, when the system is resource-constrained because there aren't enough altruistic nodes to close the gap between the supply of and demand for bandwidth, it is impossible for all nodes to download the stream with no packet loss. Under these circumstances, nodes that contribute upload bandwidth to the system are given a higher priority and tend to suffer much lower packet loss.

So far we have only investigated this algorithm under

very limited circumstances. While our initial results are promising, we still need to investigate the effect of many real-world conditions, such as the fact that node bandwidths do not fall into a small number of well-defined categories, and that available bandwidth varies over time. Moreover, we have yet to investigate the ways in which nodes may game the system.

We believe that every practically deployed peer-to-peer streaming system needs to give nodes an incentive to upload as much as they can in order to ensure that the system remains resource-rich and operates well. However, it is better to avoid shutting out nodes that are unable to upload as fast as they download unless there are insufficient altruistic nodes in the system to make up the deficit.

8 Related Work

Traditional multicast approaches have relied on building spanning trees over the network and pushing data over those trees in order to minimize delay. This creates parent-child relationships that make it hard to identify and penalize nodes that do not upload based purely on local observations.

However, Ngan, Wallach, and Druschel propose a general reputation-based system [8] to detect and penalize free-riders. Their solution can be applied to any tree-based multicast system. However, their solution require the multicast trees to be rebuilt continuously. We believe our system to be easier to implement in a decentralized manner in practice.

Levin, Sherwood and Bhattacharjee describe an interesting overlay structure[7] for file swarming that is a radical departure from BitTorrent and other tit-for-tat approaches. However, in the current form it has two severe limitations. Firstly they assume that every node in the system has exactly the same upload capacity. Secondly, the disincentive to defect comes from the *collapse of the entire system* when a single node defects. As the authors acknowledge, these drawbacks make the system impractical in its current form, but these problems may be alleviated with further research.

9 Conclusion

We argue that in order for a peer-to-peer streaming system to be robust, it is important to have an effective mechanism to give nodes an incentive to upload as much as they can. Our investigations show that naïve application of tit-for-tat mechanisms that work well in file-sharing systems do not perform satisfactorily in streaming systems due to additional bandwidth and delay constraints. We have outlined an algorithm called "Token Stealing" that runs locally on every node and relies only on direct observations without the need for network-wide or third-party coordination. Our experiments show that this algorithm helps reduce the packet loss for nodes that contribute upload bandwidth in a resource-constrained system, while not shutting nodes with poor upload capacities out of resource-rich systems.

References

[1] E. Adar and B. A. Huberman. Free Riding on Gnutella. *First Monday*, 5(10), Oct 2000.

[2] M. Castro, P. Druschel, A. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: High-Bandwidth Multicast in Cooperative Environments. In *SOSP*, 2003.

[3] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. Planetlab: an overlay testbed for broad-coverage services. *SIGCOMM Computer Communication Review*, 2003.

[4] B. Cohen. BitTorrent, 2001. http://www.bitconjurer.org/BitTorrent/.

[5] J. Jannotti, D. K. Gifford, K. L. Johnson, M. Frans Kaashoek, and J. O'Toole, Jr. Overcast: Reliable multicasting with an overlay network. In *OSDI*, 2000.

[6] D. Kostić, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: high bandwidth data dissemination using an overlay mesh. In *SOSP*, 2003.

[7] Dave Levin, Rob Sherwood, and Bobby Bhattacharjee. Fair file swarming with fox. In *Fifth International Workshop on Peer-to-Peer Systems*, 2005.

[8] T. Ngan, D. S. Wallach, and P. Druschel. Incentives-compatible Peer-to-Peer Multicast. In *Second Workshop on the Economics of Peer-to-Peer Systems*, 2004.

[9] Vinay Pai, Kapil Kumar, Karthik Tamilmani, Vinay Sambamurthy, and Alexander E. Mohr. Chainsaw: Eliminating trees from overlay multicast. In *Fourth International Workshop on Peer-to-Peer Systems*, 2004.

[10] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *ACM SIGCOMM*, 2001.

[11] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. *Proceedings of Multimedia Computing and Networking*, 2002.

[12] K. Tamilmani, V. Pai, and A. E. Mohr. SWIFT: A system with incentives for trading. In *Second Workshop on the Economics of Peer-to-Peer Systems*, 2004.