Fair Bandwidth and Storage Sharing in Peer-to-Peer Networks

Tsuen-Wan "Johnny" Ngan
twngan@cs.rice.edu

Animesh Nandi
animesh@cs.rice.edu

Atul Singh
atuls@cs.rice.edu

Department of Computer Science, Rice University

Abstract

While cooperative p2p applications are designed to share the resources of each computer in an overlay network, users do not necessarily have an incentive to donate resources to the system if they can get the system's resources for free. We study mechanisms to enforce and encourage sharing for file sharing systems. In this paper, we present two such architectures, designed for systems with different limiting resources, and discuss some problems we faced and our future directions.

1 Introduction

Resource sharing is the key for most cooperative p2p systems to be useful. Ideally, each participant in the system provides their resources in exchange for using resources from the system. In practice, however, many users may choose to consume the p2p system's resources without providing any of their own resources for the use of others [1]. These selfish behaviors hinder the p2p systems from performing optimally, and sometimes could even prevent them to function. Yet for a purely decentralized p2p system, it is hard to enforce fair sharing policies.

We consider architectures to enforce and encourage fair sharing on a p2p file sharing system like PAST [8]. We propose auditing and accounting mechanisms for systems where the limiting resource is storage space and bandwidth, respectively. Our simulation results for both architectures demonstrate that "freeloaders" are made to suffer enormously as compared to good users. We argue that these mechanisms are effective by discussing how we solve different types of potential flaws and providing answers to some concerns of the mechanisms. We also point out some of our future research directions.

For the rest of this paper, we briefly describe the architectures, and discuss how they are effective against dishonest, selfish nodes. Section 2 considers a storage-constraining design in our prior work [5] and some possible cheating behaviors. In Section 3 we propose a new design when the constrain of the system is the upload bandwidth. Finally, Section 4 concludes.

2 Storage-constraining designs

In this section, we describe a storage accounting system, targeted at p2p applications where storage space (i.e., free disk space) is the limited commodity in the system. An example of a system like this is a remote backup services, where only the writer of a given data block might want to read it. More detailed descriptions on the mechanism with simulations and an economics analysis on how global parameters of the system can be chosen can be found in previous papers [5, 4].

We assume the existence of a public key infrastructure; such an infrastructure can be provided by having a central authority assigning nodeIds. This is a reasonable assumption, given that storage services should only be provided by reliable and persistent peers. It is also imperative to ensure that nodes are actually storing the files they claim to store. This is guaranteed by a *challenge* mechanism, where each node periodically picks another node that stores the replica of a file it is storing and queries the target for a keyed hash of some randomly selected blocks of the file. Of course, all other replicas of that file should be notified in advance so that the challenger would know if there is any request for the file during the challenge, and may choose to redo the challenge.

2.1 Auditing mechanism

We consider an approach that requires nodes to maintain their own records and publish them, such that other nodes can audit those records. Every node maintains a *usage file*, digitally signed, which is available for any other node to read. The usage file has three sections: (1) the *advertised capacity* this node is providing to the system; (2) a *local list* of all files that the node is storing locally on behalf of other nodes; and (3) a *remote list* of keys of all the files published by this node (stored remotely). Together, the local and remote lists describe all the credits and debits to a node's account.

A node might fabricate the contents of its usage file to cheat the system. If it were to increase its advertised capacity beyond the amount of disk it actually has, it might

attract storage requests that it cannot honor, assuming the p2p storage system is operating at or near capacity. This is probably a safe assumption, since nodes have no incentive to provide more space than that is required by the system, costing it extra storage space and communication overhead.

To prevent fraudulent entries in either local or remote list, we allow any node to anonymously request for the usage file of any node. Anonymity can be accomplished using a technique similar to Crowds [6]. We require nodes to perform two forms of audit: (1) *normal audit* checks the remote lists of all nodes that the auditor is storing a file for; and (2) *random audit* checks a randomly chosen node and all the nodes that appeared in its local list. This ensures that a node can neither miss out a file in its remote list, nor make up unmatched entry in its local list.

We have performed simulations on the communication overhead on networks with up to 100,000 nodes, while varying the total number of files and node and file turnover rates. Auditing, even when performed periodically, is scalable with almost constant per-node bandwidth requirement, and in a variety of conditions, the overhead is quite low — only up to hundreds bps, a small fraction of a typical p2p node's bandwidth. It can be further reduced by transmitting only the diffs of usage logs for subsequent usage file requests. Detailed experimental settings and results can be found in the prior paper [5].

2.2 Cheating

Auditing allows us to discover cheating behaviors. Once a cheating node is discovered, the auditor can send the signed confession of the cheater to the centralized authority, which can then invalidate the identity of the cheater.

While the entries in local/remote lists are used to record the storage of a file, it is not mandatory for the entries to associate with files; i.e., if A puts an entry in her local list and B puts a respective entry in his remote list, everything will look legitimate to an auditor, even if no actual file is stored. However, A and B together cannot cheat for extra space, as the sum would always be even out. In this example, A must have credits to cover the debt with B, or she would be found cheating. In fact, this form of matched entries allows nodes to trade unused storage space for real world money outside the system.

Another form of attack involves forming a cheating chain or cycle. The idea is that nodes always put their debt off their own book to their successors. In case of a chain, random audit would discover a *cheating anchor* where the books did not balance. A cheating cycle attack exploits the time latency between audits, and a node puts its debt off its own book only when it is being audited. Since we assumed that only a small number of nodes will be compromised, the cheating nodes will form a small cycle. This attack can also be detected, since other nodes auditing such nodes see the recent logs of their usage files, and such form of continuous suspicious behavior could be taken to the centralized authority for arbitration.

One potential flaw of the design is that if two nodes that are responsible for storing replicas of the same file collude and only store one copy, it would not be detected by the challenge mechanism. We believe that this is an unsolvable problem, given that colluding nodes could communicate in secret channels and both will appear to have access to the file. However, since nodeIds are assigned by a centralized authority randomly, it is extremely unlikely that a malicious user can control two nodes with adjacent nodeIds, and even if they could the loss is only the reliability provided by one replica.

3 Bandwidth-constraining designs

In this section, we consider systems where bandwidth is the constrained resource. This is the case for content distribution networks, where people store popular files on a p2p storage network for its availability, geographical diversity, accumulated bandwidth, etc. For example, BitTorrent [2] facilitates large numbers of nodes all trying to acquire exactly the same file, where every BitTorrent node acquires some subset of the file and trade blocks with other nodes. We consider a more general network, and propose a simple, local accounting mechanism which involves no communication overhead.

3.1 Accounting mechanism

Every node in the system maintains two numbers on all nodes with which it has exchanged data: the number of objects (or some other globally-specified block size) sent and the number of objects received. The difference of these two numbers says something about the *debt* or *credit* that a node has with its peer. Based on this debt/credit measure, a node can decide whether to honor or refuse requests from any node.

While this debt-based mechanism ensures fair *pairwise trading* of documents with each other, the number of requests is not necessarily balanced on all pairwise relationships. If A has credit from some large number of nodes, but A wants to read an object from a node B which does not owe A anything, how can A leverage the credit it already has? We solve this problem by *transitive trading*: we find a *debt-based path* from A to B, where each node in the path is in debt to the previous node in the path. This path can be found by using debt instead of network latency as the metrics in the locality-based routing supported in some p2p substrates like Pastry [7]. Once we

have identified such a path, we could conceivably rearrange all the debts such that the B now owes something not to its predecessor in the route, but instead to A directly.

To make transitive trading work, we need a protocol that is robust against any node in the chain cheating. Rather than trying to perform some kind of complex cryptographic commitment protocol, we take an incremental approach. We divide an object into small, fixed-size blocks. A packet is routed from A through a chosen debt-based path to B, and each node in the path reduce the debt from the next hop by a block. When B receives this control packet, it knows that its debt from the previous node is reduced, and thus send the block directly to A. If any party refuses to pass along the control traffic, then the data traffic will stop as well, with the party dropping the packet getting at most one "free" block. Since nodes want to reduce their debts for its own further object requests, it is for their own good to keep the traffic alive.

Bootstrapping. When a new node joins the system, it has no debts and no credits. How can this node obtains objects to start with? We see two possible approaches. First, when a new node joins, it is required by the underlying storage system to store a replica of some files based on its nodeId, hence getting objects free of cost from its neighbors. Second, we rely on the altruism of nodes by defining a *debt threshold*, where nodes will honor the first few object requests within this threshold from any node.

Relationship throttling. Unfortunately, as the number of nodes in the p2p system grows, the amount of altruism from honest nodes could still be abused by "freeloaders". We propose a mechanism called *relationship throttling* to limit the abuse of altruism. Consider mature nodes that have been serving objects for a while. They have already accumulated enough credit for requesting objects in the future, and may not be interested in serving objects to unknown nodes. Thus, they can simply turn down any new connection. By doing so, the amount of altruism that freeloaders can abuse could be greatly reduced.

3.2 Evaluation

It is not easy to come up with an evaluation of the mechanisms, largely because there are various reasons for freeloading. For our purpose we measure the mechanisms by the running average of the number of requests (or *efforts*) for a node to retrieve its desired objects. The *quality of service* (QoS) is therefore defined to be inversely proportional to the effort.

We have performed experiments on the mechanisms. We simulate 500 nodes, with 20,000 objects distributed uniformly across the nodes; each object is replicated to a total of three nodes. These objects are all the same size, and their popularity is based on a Zipf distribution. We consider two type of nodes: a *good node* retrieves objects as well as serves objects according to the design, while *freeloaders* only retrieves objects but never serves any.

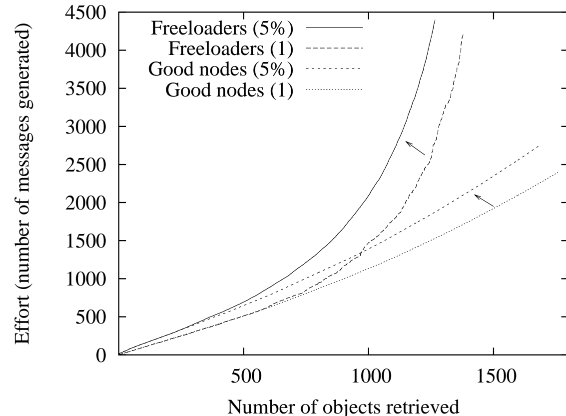

Figure 1: Effort required to fetch objects for good nodes versus freeloaders, with exactly one freeloader vs. 5% freeloaders. Arrows illustrate the effect of adding more freeloaders to the system.

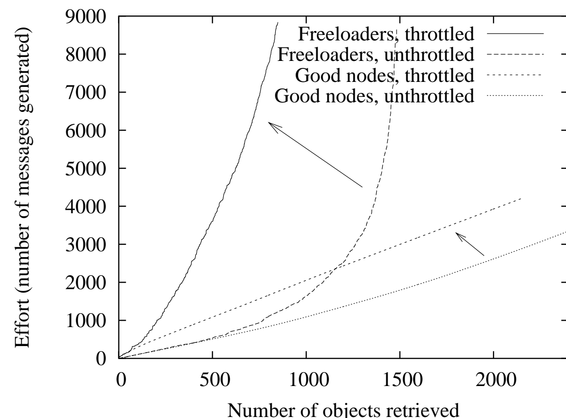

Figure 2: Effort required to fetch objects for throttled vs. unthrottled connections when there is one freeloader. Arrows illustrate the effect of enabling throttling.

Figure 1 shows that the effort required to fetch an object is the same for good nodes and freeloaders at the beginning, but after a short while the freeloaders quickly have to expend significantly more effort to acquire additional objects. Also, increasing the fraction of freeloaders affects the performance of both freeloaders and good nodes, but they suffer roughly to the same extend.

Figure 2 shows the effectiveness of relationship throttling. By introducing relationship throttling, good nodes experience small increases in the effort necessary to download desired objects, while freeloaders experience a

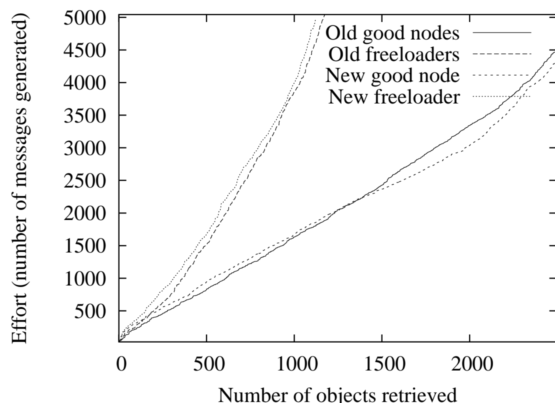

Figure 3: Effort required to fetch objects for "old" and "new" nodes. Old nodes are those present in the system from the beginning, and new nodes join the system after it has attained a steady state.

disproportionately larger increase in the effort required. Relationship throttling is an effective strategy for limiting the effectiveness of freeloaders.

One concern for relationship throttling is that nodes joining the system after it has attained its steady state may also suffer. Figure 3 shows that the required effort is not really affected by when the nodes are joined, but only whether they are freeloaders or not.

3.3 Further discussions

Although the preliminary results are encouraging, there are some interesting concerns on the design that we can address a bit further.

First, how can a mature node decide when to accept and refuse new connections? This can be simply by measuring its own recent QoS. If a node can easily retrieve objects from the system, chances are its existing relations and credits acquired are sufficient for them to find debt-based paths easily, and it does not need to improve them by accepting new connection. If its QoS is low, then it would probably want to improve it by establishing more relations.

Second, while finding a debt-based path facilitates transitive trading, a node greatly in debt with large number of nodes would receive a large number of object requests and can interpose a greater fraction of traffic. This is not an important concern because privacy on who has retrieved which object is not our design aim. There exists other systems addressing privacy, usually at the expense of performance. This is also not a denial of service attack, since nodes can always use alternate routes when one fails.

Third, we argue that nodes will correctly maintain the numbers of objects sent or received. This is because these numbers are local counters, only read by the node itself. They tell the node how other nodes think the relationship between them is. Since changing the local counters do not affect the view of other nodes, the numbers are maintained for the node's own good. Indeed, nodes are free to not maintain the numbers, but then they would have no idea on which nodes would be more likely to honor their requests.

Fourth, the mechanisms might appear to be similar to a negative reputation system, yet negative reputations would not work on systems with cheap pseudonym [3]. This is not entirely true in our design, since established relations with mature nodes is a valuable positive reputation, and that would facilitate them in seeking debt-based paths in the future.

Another concern is that the metrics effort may not be accurate to quantify the gain and loss of freeloading, because a freeloader could still end up sending much fewer data packets despite sending more requests. Another metrics could be the average time required for a freeloader to fetch an object, or other costs (e.g. CPU cycles) they need to spend. It remains unclear what metrics should be the best and how to adjust our mechanisms to make freeloaders suffer while maintaining the high QoS of honest nodes.

4 Concluding remarks

This paper presents designs for enforcing fair sharing of p2p resources. We have considered, independently, on systems where the resource constrain is on storage space and bandwidth. Our storage incentives solution, requiring nodes to perform random audits on each other's published resource usage lists, has extremely low bandwidth overhead and scales nicely to large networks. Our bandwidth incentives solution only requires nodes to track their individual pair-wise debts and uses routing, based on these debts, to guarantee a high likelihood of success in finding a node willing to transmit a desired file, while simultaneously limiting the ability of freeloaders to benefit from the altruism of good nodes. We have also discussed several concerns of the mechanisms and how we cope with those potential problems. In the future, we plan to look for better metrics in evaluating the system and then adjust our mechanisms accordingly.

Acknowledgments

This research work was jointly done with our advisors Peter Druschel and Dan S. Wallach.

References

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

[2] B. Cohen. Incentives build robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, June 2003.

[3] E. Friedman and P. Resnick. The social cost of cheap pseudonym. *Journal of Economics and Management Strategy*, 10(2):173–199, 2001.

[4] A. C. Fuqua, T.-W. J. Ngan, and D. S. Wallach. Economic behavior of peer-to-peer storage networks. In *Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, June 2003.

[5] T.-W. J. Ngan, D. S. Wallach, and P. Druschel. Enforcing fair sharing of peer-to-peer resources. In *Proc. IPTPS'03*, Berkeley, CA, Feb. 2003.

[6] M. K. Reiter and A. D. Rubin. Crowds: Anonymity for web transactions. *ACM Transactions on Information and System Security*, 1(1):66–92, 1998.

[7] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object address and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM Int'l Conf. on Distributed Systems Platforms*, Heidelberg, Germany, Nov. 2001.

[8] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. SOSP'01*, Chateau Lake Louise, Banff, Canada, Oct. 2001.