Incentives-Compatible Peer-to-Peer Multicast

Tsuen-Wan "Johnny" Ngan Dan S. Wallach Peter Druschel*
Department of Computer Science, Rice University

Abstract

Many peer-to-peer (p2p) system designs assume cooperative environments, with all clients correctly running the same software. Any client who modifies its software may be able to unfairly benefit. This paper considers such fairness issues in the context of p2p multicast streaming services. We present mechanisms that can distinguish nodes with selfish behavior and reduce the quality of service experienced by these selfish nodes from their peers. The peers make their judgments strictly by observing the behavior of their upstream peers. We only require that the multicast trees be periodically rebuilt, increasing the likelihood that a freeloading node's downstream peers will later be upstream of the freeloader and can retaliate by refusing to serve the offender.

1 Introduction

P2p multicast systems [2, 15] have demonstrated that streaming media applications can scale to reliably support large numbers of nodes without the need for the costly server and network infrastructure. Unfortunately, these systems assume that all the peers are correctly following the protocol. If a node was to refuse to transmit data to its downstream peers, or if that node was to simply refuse to accept any downstream peers, it could "freeload" on the system. If every node were to follow a similar policy, the system as a whole would collapse.

One way to solve the freeloading problem is to design *incentives-compatible* policies. We wish to build applications such that nodes maximize their utility by correctly following the prescribed protocol. A number of incentives-compatible p2p systems have been built, generally following a variety of tit-for-tat strategies. These systems are intended to provide fair sharing of disk storage [5, 17] or network bandwidth while downloading large files [3]. Multicast applications represent a related problem, but existing tit-for-tat mechanisms do not map cleanly onto multicast systems, where (relatively) static distribution trees are constructed once and used forever. We need a way to detect misbehaving peers and refuse to grant them service. This paper describes some simple mechanisms that use only first-hand observations, thus avoiding many thorny trust issues. Nodes remember when their upstream peers fail to provide them with good service. By requiring the multicast trees to be periodically rebuilt, the upstream relationships can reverse, giving nodes a chance to refuse service to the now downstream freeloaders.

Section 2 discusses the threat model and provides some background. Section 3 describes different approaches to implementing fairness policies in p2p multicast systems. We present our experimental results in Section 4. Finally, Section 5 discusses related work and Section 6 concludes.

2 Model

While the ideas in this paper are general enough to be applicable for almost any tree-based multicast systems (e.g., Bullet [15]), for concreteness we will discuss our system in terms of SplitStream [2]. SplitStream supports application-level multicast above Pastry [20], a p2p routing substrate. The key idea behind SplitStream is to split the original content stream into k stripes and to multicast each stripe using a separate multicast tree. Nodes subscribe to k different trees, with the roots spread uniformly around the Pastry ring. Every node will (most likely) be an interior node in exactly one tree and will be leaf node in the remaining $k-1$ trees. Thus, the forwarding load is distributed among all participating peers. If each node supported a fan-out to k children, then the total in-degree and out-degree would be equal.

The splitting strategy also provides robustness against packet loss. Audio and video stream types could potentially be split using media-specific codecs that allow lower-quality media streams to be partially reconstructed with only part of the data. Stripes for general-purpose data could also benefit by using error correcting codes.

Of course, this leaves room for a variety of freeloading behaviors. A node could falsely claim its outgoing bandwidth is fully utilized and refuse to accept a new child. A node could likewise accept a new child but refuse to send it any data. A node might avoid joining the one tree where it would be an interior node, only joining the $k-1$ trees where it's a leaf. Nodes might even form a conspiracy if cooperation helps them to freeload. This paper addresses such *self-interested* behaviors, but does not address *malicious* behavior, where a node's goal might be to deliberately prevent distribution of the streaming media or to otherwise damage the Pastry routing or SplitStream service. Castro et al. [1] discuss a number of techniques that might limit the damage a malicious node can cause to a p2p network; many of those ideas could be applied here.

*Email: {twngan, dwallach, druschel}@cs.rice.edu

3 Designs

In this section, we first describe a naïve approach and expose some of its problems. Then we will discuss the space of possible mechanisms that might be used to detect freeloaders and how they might be combined together to form a robust incentive-compatible policy.

3.1 A naïve approach

A selfish node can always claim that it could not receive the stream from its parent, and therefore be unable to forward the data stream. Assuming the multicast trees are always constructed in a "fair" manner according to the prescribed protocol, a naïve approach to solve this problem might be to require each node, when it fails to receive the desired data from its parent, to send (random) data of size equal to the expected stream to all its children. Since every node is required to transmit *something*, it might as well transmit the correct data.

This approach has two obvious problems. One is that it wastes bandwidth, potentially causing legitimate traffic to be dropped when the underlying network is suffering congestion. Furthermore, nothing in this approach prevents nodes from claiming to already have enough children and thus refusing to accept any more. Even in a legitimate multicast tree construction, some nodes may, depending on the protocol and by good chance, become leaves, without any requirement to retransmit content. Differentiating between good luck and freeloading will require more effort.

3.2 Fairness mechanisms

We need mechanisms that can distinguish selfish nodes from nodes that are following the protocol correctly. We wish to focus on mechanisms that individual nodes can follow, based strictly on information they observe about their peers, as well as information they might infer about nodes between themselves and the root of any given tree.

Debt maintenance When node *A* forwards a stream data packet to a node *B*, both nodes can track that *B* owes *A* a debt of one packet. If the debt exceeds some threshold, *A* might refuse to send further data to *B*.

Periodic tree reconstruction If multicast trees are constructed randomly, some nodes may be stuck in unfair or unfavorable positions if there are freeloaders. A lucky node might happen to be a leaf, where an unlucky node might happen to be downstream from a selfish node that is refusing to forward data to its children. By periodically reconstructing the multicast tree, a node will only ever benefit or suffer from such situations for at most a fixed time period. New multicast trees can be constructed concurrently while existing trees are in use. We require only that the new multicast tree be sufficiently different from the old one that a leaf node will be unlikely to have the same ancestral nodes after the old tree is replaced. There will remain a trade-off between the bandwidth overhead of tree reconstruction and the desire for smaller time steps. Smaller time steps allow nodes to respond more rapidly when they detect that a node is being selfish.

Parental availability When a node joins a multicast tree and is refused service by its prospective parent, it has no way to determine if the prospective parent is genuinely serving its maximum number of children or if it is freeloading on the system. A freeloader can always claim to be serving its conspiring peers. If service is refused once, it could just be bad luck. If, after numerous tree reconstructions, the prospective parent has demonstrated a history of refusing service to its children, then the child can legitimately refuse to serve the freeloader if and when their parental roles become reversed. The ability of a child to measure this *parental availability* will depend on the specific details of how multicast trees are constructed in any given system.

When a node joins a SplitStream tree, for example, it routes a message toward the root of that tree. The first node that receives the message is most likely to become the joining node's parent. If this node refuses the connection, saying it has enough children already, the joining node must search for another parent. It will first search the children of the failed parent, and then its siblings and grandparent, recursively. If SplitStream nodes are operating correctly, these searches will be unlikely to occur, and service will most likely be found with one of the failed parent's immediate children. As such, any parent that consistently refuses to accept a node as a child is highly likely to be a freeloader.

Reciprocal requests Two well-behaved nodes would be expected to have an equal chance of being parent or child in any given multicast tree. A freeloader, however, might regularly refuse to accept children. When the freeloader *A* asks some prospective node *B* to be its parent, *B* needs a way to judge whether *A* has had a history of behaving selfishly. To address this, we allow *B* to break the traditional join protocol and instead occasionally attempt to make *A* its parent by requesting to join directly under *A* for a multicast tree where *A* is supposed to be an interior node. This can be done whenever the number of recent requests from one direction exceeds a constant factor more than requests in the opposite direction. This would allow *B* to determine whether *A* is misbehaving, and thus have a stronger basis for ignoring *A* in the future.

Ancestor rating Another approach, an extension of debt maintenance, is to apply debts and credits not only to a node's immediate parent, but to all of their ancestors who should have been responsible for forwarding data from the multicast root. Whenever a node receives a packet, it increments its confidence value of each node in the path to the root. Whenever an expected packet is not received (this can be noticed if the packets should arrive at a timely, periodic fashion, as in video and audio streams), the node decrements the confidence value of each node in the path to the

root, blaming them all equally, for the lack of any more specific information. When the trees are reconstructed, any blame assigned falsely or due to lost packets would average out as nodes are later observed to behave correctly. Freeloading nodes, on the other hand, would be consistently blamed for their misbehavior. Service would eventually be refused to these freeloaders.

3.3 Authenticity of data and path

Our mechanisms rely on the knowledge of ancestors. A selfish node, of course, has no incentive to provide such information correctly. False information might allow good nodes to be falsely considered to be freeloaders; likewise, false information might allow freeloading nodes to escape detection. Here, we outline a low-cost method to authenticate the stream data and verify the integrity of the path. We borrow ideas from hash chains [19] and path authentication in Ariadne [14].

First, the source creates a hash chain by randomly generating a value x_n (for sufficiently large n), and iteratively computing $x_{n-1},\ldots,x_0$ by $x_i = h(x_{i+1})$ with a cryptographically secure one-way hash function h (e.g., MD5 or SHA-1). One important property of one-way hash functions is that while it is cheap to compute a hash, it is computationally infeasible to find its inverse. Thus, given x_{i+1}, it is trivial to verify that it hashes to x_i, but it is infeasible to find x_{i+1} from x_i. We assume that the source can distribute x_0 as an initial shared secret to all receivers.

Before the source sends the i^{th} packet, it computes the base message digest $d_i = h(\text{data}_i, x_i)$. Whenever a node sends the packet, it hashes the message digest it receives from its parent (or d_i for the source) with the receiving node's nodeId. Thus, the message digest received by the source's child A would be $h(d_i, A)$ and that by A's child B would be $h(h(d_i, A), B)$ and so on. Each packet will also include the hash chain value used in the previous packet, i.e., the $i+1^{\text{th}}$ packet contains x_i. Upon receipt of x_i, each node can confirm that $x_{i-1} = h(x_i)$. Each node can then verify the integrity of the previous packet by reconstructing the message digest using x_i and the path.

In case of lost packets, a node only needs to hash the value multiple times until it matches the last seen x_i. Likewise, a node joining an ongoing streaming session only needs to hash the value multiple times until it matches x_0. If the source ever runs out of all the values in the hash chain, it can generate a new chain on the fly and use the old chain to authenticate the new one. Each multicast tree can use a separate hash chain so that other trees can still be use while one is under reconstruction.

Under this scheme, nodes cannot fake the path from the root to their children without knowing x_i, which would not be revealed until after the packet becomes obsolete. Nodes can still lie about their children, however. If that becomes an issue, we can require nodes to sign lists of their children, creating a structure analogous to a Merkle hash tree [16].

3.4 Sybil attacks

The rating mechanisms described above can all be potentially defeated if nodes with poor reputations can quit the system and rejoin under new identities, an example of a Sybil attack [8]. While we could address these attacks by requiring certified nodeIds [1], we can also limit the effectiveness of such attacks by putting new nodes through a probation where they experience a lower quality of service. In SplitStream, where there are k trees being used concurrently, we might reconstruct one tree for each time step. If we close these trees to new members after they start running, then a new node will not be able to join a tree until it is being reconstructed, and will not receive all k streams until k time steps have elapsed. Thus, it will get a lower quality of service when it first joins, with its quality progressively improving over time. This may or may not be an inconvenience to legitimate nodes. If, for example, nodes are subscribing to a lecture that starts at a known time, they would need to join k time steps in advance. If a time step was 15 seconds and $k = 16$, then the probationary period would only be four minutes long.

A selfish node might attempt to join under multiple identities with the hope of getting some portion of the stream with each pseudonym. Regardless of the pseudonyms, the selfish node will be immediately required to participate in the protocol and will suffer if it freeloads. Furthermore, a node using multiple pseudonyms will pay some fixed overhead in the underlying p2p protocol for maintaining each pseudonym, thus providing a disincentive to creating such pseudonyms. As a result, nodes have an incentive to join under a single identity and to behave correctly, allowing them to develop a positive reputation.

4 Experiments

In this section, we use simulations to study the effectiveness of a variety of different mechanisms. We study several mechanisms in isolation and then describe a combination that is more effective at discriminating freeloaders from normal nodes. All experiments are run on an instrumented version of SplitStream using 500 nodes with randomly chosen nodeIds. Since SplitStream considers node proximity when building multicast trees, node "locations" are randomly distributed on a plane, with proximity between two nodes determined by their Euclidean distance. Each node attempts to subscribe to $k = 16$ trees, and will accept up to 16 children. The root node transmits one "data unit" to each multicast tree and then all trees are reconstructed at every time step. (For an actual implementation trying to spread the load of tree reconstruction, we might cut one time step into 16 smaller steps and reconstruct one of the 16 trees per step. The net cost would be the same.)

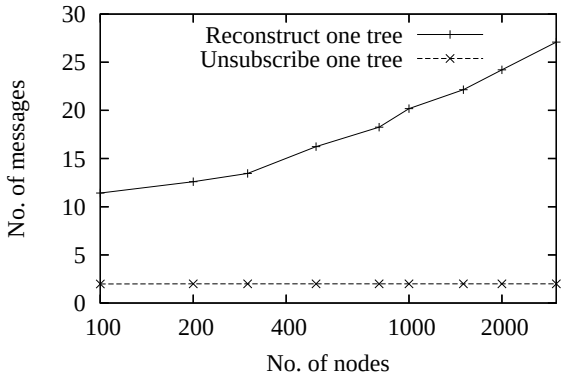

Figure 1: Average tree reconstruction cost.

4.1 Tree reconstruction cost

Tree reconstruction would not be useful if the cost was prohibitively expensive. We first study the cost of reconstructing and discarding trees. Figure 1 shows the average number of messages sent by every node in order to reconstruct a tree. Since subscribing to a tree is simply sending a subscribe message to a specific nodeId, the cost is proportional to the log of the number of nodes. As each message is very small in size (it contains a treeId and a few nodeIds), only a few kilobytes are transmitted by each node, which is minimal relative to typical data rates for streaming video. To unsubscribe from a tree, a node only needs to notify its parent in the tree, therefore the cost is constant regardless of the size of the system. Moreover, this unsubscription cost can be saved if all the nodes in the tree discard the tree at the same time. This is possible if the data source can include this information in the data stream.

To estimate the overhead in practice, consider video streaming to 500 nodes. Assume that the video is streaming at 128Kbps, the typical upstream bandwidth for a DSL user. Figure 1 shows that on average each node needs to send 16 messages to reconstruct one tree. Assume that each message is of size 128 bytes and all 16 multicast trees are reconstructed every two minutes. The total overhead would only be 1.71% of the stream.

4.2 Debt

Consider two randomly chosen nodes in the SplitStream system. If the trees are constructed randomly, the odds of one node being the parent or the child of the other are the same as a random coin flip. As trees are reconstructed, the expected average debt that might be accumulated will tend to vary with the square root of the number of rounds [13]. We can thus define the *debt level*:

$$\text{Debt level} = \frac{\text{accumulated debts/credits}}{\sqrt{\text{total transfers}}} \quad .$$

Figure 2 shows the cumulative distributions of debt levels, with 5% selfish nodes after 256 rounds of tree reconstructions. Despite the tree reconstructions, we see that debt levels do not discriminate well between selfish and normal nodes. We believe this occurs mainly as a result of the SplitStream's preference for routing to "local" nodes, meaning that many nodes will have the same pairings, round after round, and other nodes will not learn enough to distinguish selfish from normal nodes.

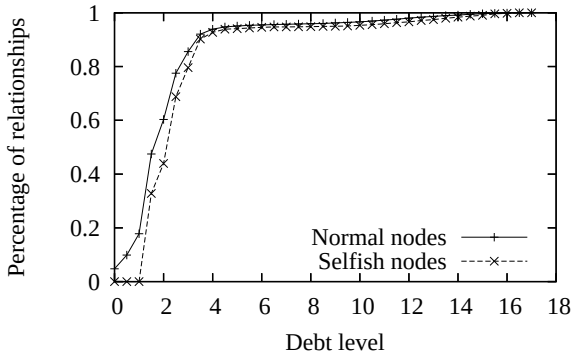

Figure 2: Cumulative distribution for debt level.

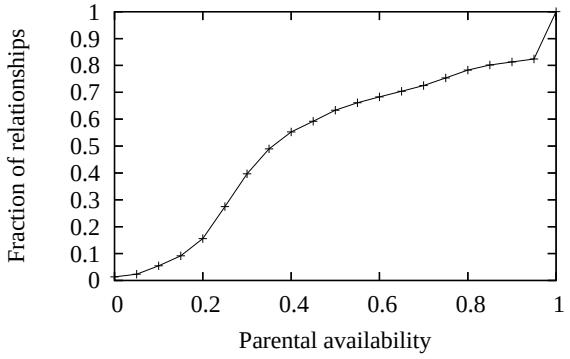

Figure 3: Cumulative distribution of parental availability when all nodes are normal.

4.3 Parental availability

In order to understand parental availability (a child's rating of how likely a given parent was to accept it as a child), we simulated a network with no freeloaders; each node will accept up to 16 children. Figure 3 shows the distribution after 256 tree reconstruction time steps. Half of the child-parent availability ratings in the system are below 0.37 and half are above. If a node was a freeloader, its parent availability rating would be zero. If we choose to cut off parents with low availability, we must take care to avoid false positives, particularly given that many legitimate parents have low ratings. For example, a cut-off of 0.44 might normally reject 58% of the legitimate parents.

4.4 Confidence

Since debts between peers and parental availability rates are insufficient, by themselves, to detect freeloading nodes, we will consider a rating mechanism (see Section 3.2) that

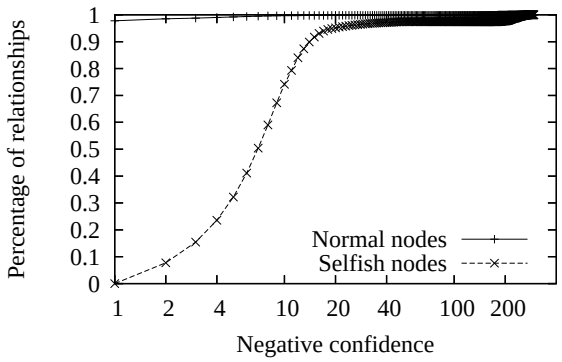

Figure 4: Cumulative distribution for negative confidence.

blames all upstream peers for any transmission failure. Figure 4 shows the distribution of negative confidence, with 5% selfish nodes, after 256 full reconstructions. Unlike debt, the confidence value can effectively distinguish selfish nodes. For example, by setting a threshold of 2, a selfish node can be positively identified by more than 90% of nodes in the system with only 1% false positives.

4.5 Refusing service to freeloaders

In this experiment, we evaluate the effectiveness of a combination of our mechanisms. We simulate a system consisting of 496 nodes correctly following our protocol and four selfish nodes. Two of these selfish nodes begin cheating immediately while the other two start cheating only after time 32. Two cheaters will refuse to forward traffic to their children and the other two will refuse to be a parent. When performing our simulations, we tried a variety of different parameters, eventually settling on the ones described here. Normal nodes will always forward data to each of their children, unless the child has:

- a confidence value of less than -2; or
- a parental availability of less than 0.44 as well as a confidence value of less than 0.2.

When accepting children, a parent can preempt its previously-accepted children for any other node with at least 0.1 higher in parental availability. Also, reciprocal requests are used when a prospective child has attempted to contact a parent at least a factor of 8 times more often than when their roles are reversed. Furthermore, we decay positive confidence values over time, multiplying them by 0.9 after each time step. As a result, nodes will forget how good their parents have been, but they will remember how bad their parents have been. A parent is thus forced to continue providing service to maintain its children's confidence; likewise, freeloaders will be forced to provide service if they ever wish to reestablish their reputation.

Figure 5 shows that our hybrid policy, considering confidence values and parental availability, effectively punishes nodes who refuse to forward traffic to their children. Nodes

Type	Count	Description
+	496	Normal nodes
×	1	Refuse to accept children after 32
⊡	1	Always refuse to accept children
⊙	1	Refuse to forward data after 32
■	1	Always refuse to forward data

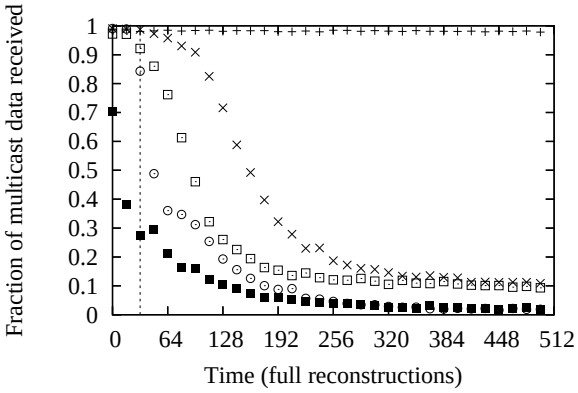

Figure 5: Fraction of multicast streams successfully received when nodes refuse to send data to freeloaders.

who refuse to accept children likewise manage to only receive 10% of the multicast data. It can also be observed that initial cooperation followed by freeloading behavior has only a limited effect in the short term and no effect in the long term.

We also performed an experiment with ten freeloaders of both types (refusing to send data to children and refusing to accept children). These freeloaders experienced similar levels of reception as the freeloaders in Figure 5, but normal nodes' reception levels dropped from 98% to 92%. This indicates that increasing numbers of freeloaders will cause worsening performance for normal nodes, but that the freeloaders will not benefit from their increased presence. For contrast, we also ran a similar simulation against the original SplitStream, with a random distribution of freeloading nodes and found that normal nodes' reception levels was 90.6%. This shows that our techniques represent an improvement over the original system, although some bandwidth is still being wasted on freeloaders. One possible way to address this might be to use a data encoding that somehow requires a node to receive above a certain fraction of the multicast data to decode anything at all. Such a scheme would reduce the utility experienced by freeloaders; if the freeloaders gain no benefit from staying, they would leave and the bandwidth they were consuming would revert back to being used by good nodes.

5 Related work

Distributed mechanisms Ngan et al. [17] consider a p2p storage system, and propose an auditing mechanism so that cheaters can be discovered and evicted from the system. Fuqua et al. [12] modeled the utility function of nodes in such a system, showing that nodes with similar prefer-

ences will have an incentive to cluster together and to reveal their preferences truthfully. Feigenbaum et al. [9] consider multicast transmissions using micro-payments, and proved the strategy-proof property of a simple cost-sharing mechanism. All such systems are examples of problems in distributed algorithmic mechanism design (DAMD) [10].

Nicolosi and Mazières [18] propose a technique for the sender of multicast data to confirm message delivery to all receivers. While this can help the sender to learn the identity of nodes refusing to forward data, it does not prevent nodes from refusing to accept children.

Reputation systems Many systems depend on nodes observing the behavior of their peers and gossiping with each other about their observations. Dingledine et al. [7] surveys many such schemes for tracking nodes' reputations.

In reputation systems, if obtaining a new identity is cheap, negative reputations can be shed easily. Friedman and Resnick [11] study the case of cheap pseudonyms, and argue that suspicion of strangers is costly. Distributed reputation systems have been proposed in a number of contexts, including MIX-Nets [6] and Gnutella [4].

Our system uses the notion of a probationary period, where new nodes see degraded service, yet must participate fully in the protocol. A similar concept appears in Tangler [21].

6 Conclusions

We have demonstrated that, by regularly rebuilding multicast trees and having nodes only track their first-hand observed behavior of their peers, freeloaders would be suitably denied service. The network and computational overhead of our mechanism is low and thus could scale to large number of nodes. It remains future work to study whether we can improve the robustness of the system to tolerate a larger fraction of freeloaders and freeloaders operating in concert with one another. In addition, the effectiveness of our mechanism may depend on the choice of multicast applications, p2p routing substrates, and network topologies. Regardless, we have shown the effectiveness of combining a node's direct observations with mechanisms to guarantee that parent-child relationships have a good chance of being reversed are effective at providing disincentives to freeloading behaviors.

References

[1] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Security for structured peer-to-peer overlay networks. In *Proc. OSDI'02*, Boston, MA, Dec. 2002.

[2] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: High-bandwidth multicast in cooperative environments. In *Proc. SOSP'03*, Bolton Landing, NY, Oct. 2003.

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

[4] F. Cornelli, E. Damiani, S. D. C. di Vimercati, S. Paraboshi, and P. Samarati. Choosing reputable servents in a p2p network. In *Proc. 11th Int'l WWW Conf.*, Honolulu, Hawaii, May 2002.

[5] L. P. Cox and B. D. Noble. Samsara: Honor among thieves in peer-to-peer storage. In *Proc. SOSP'03*, Bolton Landing, NY, Oct. 2003.

[6] R. Dingledine, M. J. Freedman, D. Hopwood, and D. Molnar. A reputation system to increase MIX-Net reliability. In *Proc. 4th Int'l Workshop on Information Hiding*, Pittsburgh, PA, Apr. 2001.

[7] R. Dingledine, M. J. Freedman, and D. Molnar. Accountability. In A. Oram, editor, *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*, chapter 16. O'Reilly & Associates, 2001.

[8] J. R. Douceur. The Sybil attack. In *Proc. IPTPS'02*, Cambridge, MA, Mar. 2002.

[9] J. Feigenbaum, C. Papadimitriou, and S. Shenker. Sharing the cost of multicast transmissions. *Journal of Computer and System Sciences*, 63(1), Aug. 2001.

[10] J. Feigenbaum and S. Shenker. Distributed algorithmic mechanism design: Recent results and future directions. In *Proc. 6th Int'l Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, pages 1–13, Atlanta, GA, Sept. 2002.

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

[12] 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.

[13] M. B. Handelsman. Distributing "heads" minus "tails". *The College Mathematics Journal*, 22:444–446, 1991.

[14] Y.-C. Hu, A. Perrig, and D. B. Johnson. Ariadne: A secure on-demand routing protocol for ad hoc networks. In *Proc. MobiCom'02*, Atlanta, GA, Sept. 2002.

[15] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *Proc. SOSP'03*, Bolton Landing, NY, Oct. 2003.

[16] R. Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology—CRYPTO'87 (LNCS, vol. 293)*, 1987.

[17] 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.

[18] A. Nicolosi and D. Mazières. Secure acknowledgment of multicast messages in open peer-to-peer networks. In *Proc. IPTPS'04*, San Diego, CA, Feb. 2004.

[19] A. Perrig, R. Canetti, D. Song, and J. D. Tygar. Efficient and secure source authentication for multicast. In *Proc. NDSS'01*, San Diego, CA, Feb. 2001.

[20] 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.

[21] M. Waldman and D. Mazières. Tangler: A censorship-resistant publishing system based on document entanglements. In *Proc. 8th ACM Conf. on Computer and Communications Security*, Philadelphia, PA, Nov. 2001.