# SLIC: A Selfish Link-based Incentive Mechanism for Unstructured Peer-to-Peer Networks

Qixiang Sun Hector Garcia-Molina
Stanford University
{qsun, hector}@cs.stanford.edu

## Abstract

Most Peer-to-Peer (P2P) systems assume that all peers are cooperating for the benefit of the community. However in practice, there is a significant portion of peers who leech resources from the system without contributing any in return. In this paper, we propose a simple Selfish Link-based InCentive (SLIC) mechanism for unstructured P2P file sharing systems to create an incentive structure where in exchange for better service, peers are encouraged to share more data, give more capacity to handle other peers' queries, and establish more connections to improve the P2P overlay network. Our SLIC algorithm does not require nodes to be altruistic and does not rely on third parties to provide accurate information about other peers. We demonstrate, through simulation, that SLIC's locally selfish and greedy approach is sufficient for the system to evolve into a "good" state.

## 1 Introduction

Peer-to-Peer (P2P) file-sharing systems organize users into an overlay network to facilitate the exchange of data. However, current deployed systems lack any "viable" incentive structures for encouraging users to behave in the best interest of the community. As a result, various forms of abuse and attack have been observed in practice. The most common ones are free loaders [2] and denial-of-service (DOS) attacks. A *free loader* is a user who only downloads files from a P2P network while never sharing any files. These users become leeches and drain resources from the community. Unlike free loaders who only reduces the resource in the network, there are also malicious users who launch active DOS attacks against P2P overlay networks. These attacks typically take the form of serving bogus files or straining the network by flooding bogus queries.

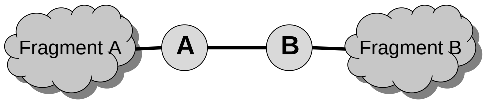

**Figure 1: Example of mutual access control**

Some researchers have proposed a global reputation-based system for combating these problems [8, 1, 5, 6, 10]. In such a system, each user is assigned a reputation by the community that reflects its contribution to and its participation in the community. This reputation can then be used to filter out free loaders or malicious users. Others have suggested imposing a bartering economy [11, 13] on the entire community where users exchange services. The economy can be implemented as micro-payments or IOU certificates.

In contrast to building a global system with reputations or economics that requires users to cooperate or some central authorities, this paper proposes a simple alternative solution SLIC (Selfish Link-based InCentive) where each user can, and in fact is encouraged to, act selfishly and greedily. We will show that SLIC allows an unstructured P2P overlay network, such as Gnutella [7], to "evolve" into a state where free loaders and attackers are ostracized quickly without asking any user to behave against his best interest.

[0]This work is supported in part by NSF Graduate Fellowship and NSF Grant IIS-9817799.

SLIC operates by taking advantage of one key property of the flooding-based search mechanism used in unstructured P2P overlay network. In flooding, when a node wants to find a particular piece of data, it sends a search query to all its neighbors on the P2P overlay network. Its neighbors then in turn forwards the search query to their neighbors, and so forth. Observe that this flooding-based search mechanism allows neighboring nodes to *control each other's access* to the rest of the network. For example, consider two nodes $A$ and $B$ in Figure 1 that share an overlay link. In order for node $A$'s queries to reach other nodes in the network fragment $B$, node $B$ must forward $A$'s queries. Similarly, node $B$ can only reach nodes in network fragment $A$ if node $A$ forwards its queries.

SLIC exploits this relationship by allowing each node to "rate" its neighbors and to use the ratings to control how many queries from each neighbor to process and to forward on. Intuitively, for node $A$ in Figure 1, if $B$ is providing great service either directly by $B$ itself or indirectly by nodes reachable via $B$ in the fragment $B$, then $A$ would give a high rating to $B$ and process and forward more queries from $B$. Conversely, bad service would reduce rating and the number of queries serviced. In other words, SLIC uses this mutual access control relationship as a means of retaliation if a node does not play fair or connects to nodes that do not play fair. To improve their service, nodes are incentivized to provide content and/or to connect to nodes that provide content.

Our simple approach has two significant advantages: (1) each user is greedy in that he is trying to maximize his own advantage in getting better service; (2) each user only keeps statistics about its neighbors and does not rely upon a trusted authority or others to give accurate "reputations" about unknown users. For the remainder of this paper, we will develop this intuition into our selfish link-based incentive mechanism (SLIC). We will show that SLIC is effective in controlling free-loaders and malicious nodes. Our main contributions are

- propose the SLIC algorithm that is used by each node to manipulate the rating (Section 2)
- illustrate that the incentive structure from nodes executing SLIC does the "right thing" (Section 3)
- show that SLIC can respond quickly when the overlay network is dynamically changing (Section 4)

## 2 Basic Algorithm

Informally, SLIC is a general algorithm that operates in periods, e.g., every minute. During each period, a node has certain capacity that it is willing to use for servicing queries from neighboring nodes on the P2P overlay. To distinguish good neighbors from bad ones, a node $u$ maintains a weight $W(u, v)$ for each neighbor $v$, where $0 \leq W(u, v) \leq 1$. A weight of 1 indicates an excellent neighbor while a weight of 0 implies a useless one. With these weights, a node $u$ then allocates its capacity to service incoming queries from its neighbors proportionally to the weights. For instance, if $u$ has two links to nodes $x$ and $y$ with weights 1 and 0.5 respectively, then in this period node $u$ will give $\frac{2}{3}$ of its capacity to queries from node $x$ and $\frac{1}{3}$ of its capacity to queries from node $y$.

At the end of a period, each node reevaluates its opinion, or weights, of its neighbors based on how much service the neighbors had provided during the current period. In particular, we measure the number of query hits that a node has received from each neighbor. A *hit* in this case means a piece of data that satisfies a search query. If a neighbor gave more service than previously expected, then the corresponding weight will increase. Similarly, less service will result in lower weight. Since quality of service may fluctuate frequently, SLIC uses an exponential decay mechanism for updating weights. Specifically, if $W(u, v)$ denotes the weight used in the previous period, $W'(u, v)$ denotes the new weight for the next period, and $I(u, v)$ denotes the quality of service from neighbor $v$ during this period, then $W'(u, v) = \alpha W(u, v) + (1 - \alpha) I(u, v)$ for some $\alpha$ where $0 < \alpha < 1$.

The weight adjustment and the capacity allocation in the SLIC algorithm create a feedback system. In other words, if node $u$ is receiving most of its query hits from node $v$, then $u$ will reciprocate by increasing the weight $W(u, v)$ and reducing other weights. As a result, node $u$ will give most of its spare capacity to handle queries from $v$. In this section, we first introduce a simple model and some notation for formally describing the algorithm in the context of an unstructured P2P file sharing system. We then give the details of the SLIC algorithm stated above. We finish by illustrating how SLIC works through two examples.

## 2.1 A Simple Model and Notation

We use a very simple model to capture the key characteristics of an unstructured P2P file sharing system that are relevant to our study and simulation. We model the overlay network as a graph $G = (V, E)$. The vertex set $V$ represents nodes (users) in the network. The edge set $E$ represents the overlay links.

We model the periodic behavior of SLIC as nodes operating in rounds; although the rounds do not have to be synchronized, for simplicity, we will assume synchrony in this paper. During a single round, each node $u$ can handle up to $C_u$ queries. This capacity $C_u$ is divided between generating new queries and answering queries from the neighbors. Thus a node generating too many queries will have little capacity to process and forward queries from its neighbors, and vice versa. We use $\rho_u$, where $0 \leq \rho_u \leq 1$, to denote the fraction of capacity used by node $u$ to generate new queries. Thus a malicious node that floods the network with queries is captured by a node with $\rho = 1$. Here we have assumed that each node can generate $\rho_u \cdot C_u$ queries each round. In practice where the P2P overlay network consists of super-nodes[1], there should be enough new queries from clients of a super-node. Moreover, even if a node generates fewer new queries, it still can use the extra capacity to service neighbors' queries. If a node receives more than $C_u$ queries from its neighbors, it can choose which $C_u$ queries to accept. When choosing queries from a single link, we assume nodes will always prefer queries with high time-to-live (TTL). (Previous work in [12] showed that preferring high TTL queries result in the most queries processed.)

We model the flooding-based search mechanism as a simple forwarding step where each node $u$ sends its $C_u$ queries chosen during round $i$ to its neighbors for processing during round $i + 1$. In the process of this forwarding step, each query's TTL is decremented. If a query has TTL 0, it is removed from the system.

We also model the amount of data that a node $v$ is sharing by a parameter we call the *answering power* $A_v$ where $0 \leq A_v \leq 1$. This *answering power* represents the probability of node $v$ having a hit that satisfies a query. In other words, a large $A_v$ value means node $v$ is sharing many files. Similarly, a low $A_v$ value (e.g., 0) represents a free loader. Note that we have simplified the answering power by assuming that a node is equally likely to have a hit for any queries, thus we are ignoring clustering effects in data shared by users. We also assume that each node can only contribute zero or one hit.

[1]A super-node is a high capacity node that acts as a proxy for a large number of slower or low capacity nodes.

In this model, we have only captured node capacity, query generation, query propagation, and likelihood of having a hit at individual nodes. We are not modeling the actual file downloads. Section 5 briefly addresses how to generalize our SLIC algorithm for handling file downloads. We also use the following notation in describing our algorithm.

- $E_u$ denotes the set of edges from node $u$.
- $W_i(u, v)$ denotes the weight of the link $(u, v)$ in round $i$.
- $Q_i(u)$ denotes the set of queries initiated by $u$ whose TTLs have expired during round $i$. For our simple model, nodes can determine exactly which queries have expired by the round number. In practice, we can use a fixed time-out for this purpose.
- $q_i(u, v)$ denotes the number of hits for query $q$ that were received from the link $(u, v)$ by round $i$.
- $\tau$ denotes the maximum TTL for each query.
- $\alpha$ denotes the exponential decay rate, e.g., $0.9$.
- $P_i(u, v)$ denote the number of queries generated by node $v$ that node $u$ decide to process in round $i$.
- $G_i(u)$ denote the number of queries node $u$ generated in round $i$.

## 2.2 Description of the Algorithm

The SLIC algorithm has two components: (1) how nodes operate while using the weights, and (2) how nodes update the weights each round. Figure 2 provides the pseudo-code for how nodes operate when running SLIC. When the system first starts, all weights $W_0(u, v)$ are initialized to 1. Subsequently, during each round, nodes first use some of their capacity to generate new queries as in step 1. The number of new queries is controlled by the parameter $\rho_u$. As mentioned previously, a malicious node that floods

```
During round $i$, node $u$ performs the follow actions:
 1: generate $\rho_u \cdot C$ new queries
 2: $W_{total} = \sum_{(u,v)\in E_u} W_i(u,v)$
 3: for each edge $(u,v) \in E_u$ do
 4:     Process and forward $(1-\rho_u)C \cdot \frac{W_i(u,v)}{W_{total}}$ queries
        with the highest TTL from the link $(u,v)$
 5: end for
 6: if there are still spare capacity, repeat steps 2 through
    5 to divide the spare capacity.
 7: Tally number of hits for newly expired queries generated by $u$, i.e., $Q_i(u)$.
 8: for each edge $(u,v) \in E_u$ do
 9:     $W_{i+1}(u,v) = compute\_weight(W_i(u,v), Q_i(u))$
10: end for
```

**Figure 2: Pseudo-code for node operation.**

```
Procedure $compute\_weight(W(u,v), Q_i(u))$
1: for each edge $(u,x) \in E_u$ do
2:     $I(u,x) = \sum_{q_i \in Q_i(u)} \frac{q(u,x)}{\sum_{(u,y)\in E_u} q_i(u,y)}$
3: end for
4: $I_{max} = \max\{I(u,x) | (u,x) \in E_u\}$
5: return $\alpha \cdot W(u,v) + (1-\alpha)\frac{I(u,v)}{I_{max}}$
```

**Figure 3: Pseudo-code for computing new weight**

the network with queries is equivalent to having a large $\rho_u$ value.

Once nodes have generated their new queries, in steps 2 through 5, each node divides the remaining capacities proportionally, according to the weights, among its links to process remote queries from neighbors. Note, however, that it is possible for a node to still have spare capacity after steps 2 through 5. For example, suppose the weights dictate that node $u$ should choose 100 queries from link $(u,v)$. If node $v$ only sends 50 queries, then there would be an unused capacity of 50 queries. In the rare event of having unused capacity, we reallocate the capacity among the remaining links as in step 6.

After nodes have chosen which queries to process and forward, in steps 7 through 10, each node $u$ then considers its own queries $Q_i(u)$ whose TTLs have expired at the current round $i$. Node $u$ uses the statistics on how many hits for the queries in $Q_i(u)$ were received from each link to update the link weights.

```
Procedure $compute\_weight\_scaled(W(u,v), Q_i(u))$
1: for each edge $(u,x) \in E_u$ do
2:     $I(u,x) = \sum_{q \in Q_i(u)} \frac{q(u,x)}{\sum_{(u,y)\in E_u} q(u,y)}$
3: end for
4: $Imax = \max\{I(u,x) | (u,x) \in E_u\}$.
5: if $P_i(u,v) > G_i(u)$ then
6:     return $\alpha \cdot W(u,v) + (1-\alpha)\frac{G_i(u)}{G_i(v)}\frac{I(u,v)}{Imax}$
7: else
8:     return $\alpha \cdot W(u,v) + (1-\alpha)\frac{I(u,v)}{Imax}$
9: end if
```

**Figure 4: Pseudo-code for computing new weight with excess scaling**

There are many ways to perform this weight update. Figure 3 shows the pseudo-code for one such update procedure. In this case, for each query $q \in Q_i(u)$, we first determine the fraction of hits contributed by a particular neighbor $v$. The contribution of a link $(u,v)$ in this round $I(u,v)$ is then simply the sum of these fractions over all queries in $Q_i(u)$ as in step 2. Once we have the contribution for each link, we find the maximum contribution by any link $I_{max}$ in step 4. We finally compute the new weight in step 5 using an exponential decay rate of $\alpha$ with the new contribution $I(u,v)$ normalized by the maximum $I_{max}$. Besides the weight adjustment shown in Figure 3, we also tried computing $I(u,v)$ as the raw number of hits or the number of queries with at least one hit. Both variations yield similar results.

The *compute_weight* procedure in Figure 3 has one weakness. Suppose that according to node $u$'s weights, node $u$ decides to process $c$ queries from its link with a neighboring node $v$. Also suppose that all nodes are generating $\frac{c}{2}$ queries per round. Now node $v$ *could* take advantage of the situation by generating more of its own queries (say $\frac{3c}{4}$ queries) rather than forwarding its neighbors' queries, thus getting more hits for its queries. This exploitation is possible because node $v$ can continue to process the same number of queries from node $u$ as before

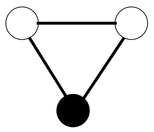

**Figure 5: Example network of $2$ white nodes with $\rho = 0.5$ and black node with $\rho = 0.9$**

to maintain its weight with $u$. This situation is undesirable because in acting selfishly, all nodes may decide to generate more queries, which cause the system to operate in a less efficient manner. We will illustrate this weakness in detail in Section 3.3. To combat this undesirable behavior, we introduce a modification called *excess scaling* whose pseudo code is shown in Figure 4.

Under excess scaling, node $u$ penalizes a neighboring node $v$ if $u$ is processing more queries from $v$ than the number of queries $u$ generated itself, i.e., $P_i(u, v) > G_i(u)$. For instance, if $u$ is generating 10 queries per round, but happens to process $20$ queries from $v$ per round, then $u$ should penalize $v$. The penalty depends on how many queries were generated by the neighbor, and is captured in step 6 in Figure 4.

As we will see in Section 3.3, this modification is sufficient to discourage nodes from generating more queries than the system norm.

During our SLIC evaluation, we noticed that the per round contribution $I(u, v)$, as computed in step 2 of *compute_weight*, is very noisy because of the stochastic nature of the number of hits a node provides. To reduce noise, instead of using the single round contribution, we keep a moving window of 10 rounds and use the average of the contributions in this window.

## 2.3 A Simple Example

To illustrate how weights change over time when running SLIC, let us begin with a simple example of three nodes as shown in Figure 5. The three nodes are connected in a ring. The two white nodes are using $50\%$ of their capacity to generate new queries, whereas the black node is trying to get extra service by using $90\%$ of its capacity to generate new queries. All three nodes have an answering power of 1, i.e., every query will have a hit at each node.

Intuitively, we hope that when running SLIC, the two white nodes will detect that the black node is dedicating

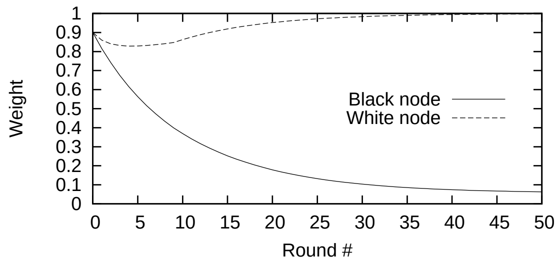

**Figure 6: Weight adjustments by the white node**

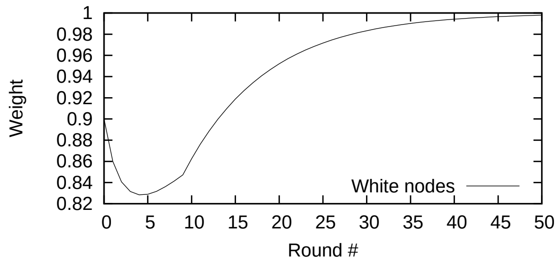

**Figure 7: Weight adjustments by the black node**

fewer resource to process their queries and retaliate by reducing their service for the black node's queries. And indeed, this situation does occur. Figure 6 shows how a white node adjust its weights as a function time. The dashed curve corresponds to the link to the other white node, and the solid curve represent the link to the black node. The x-axis gives the round number. The y-axis shows the weight. As expected, the white node quickly reduces its weight for the black node while boosts the weight for the other white node. It is also interesting to note that the weight for the black node does not drop to $0$, as it stabilizes around $0.05$. The reason is that although the black node is generating more queries, it is still providing some service with its $10\%$ spare capacity, thus getting a small amount of service in return.

For completeness, we show how the black node adjusts its weight for the white nodes in Figure 7. Since it is getting the same kind of service from both white nodes, the weights are boosted to 1. Note that in both Figures 6 and 7, the weights initially drops before reaching 1. This drop is the side effect of our implementation that uses a moving window of size 10 for computing the per-round contribution. When the simulation starts initially, the first round

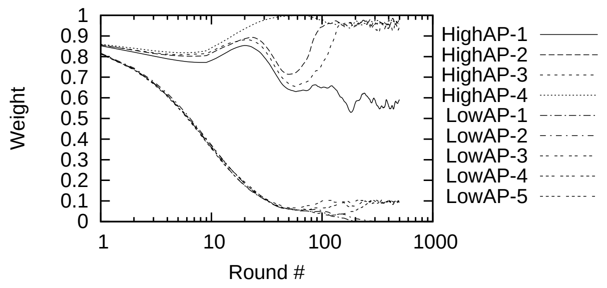

**Figure 8: Illustration of weights changing over time for a complete graph of $10$ nodes.**

has a zero contribution because no queries has been propagated yet. The effect of this initial zero lingers around until it expires from the moving window.

## 2.4 A More Involved Example

Now consider a fully connected graph of $10$ nodes where each node has $\rho = 0.2$, i.e., devotes $80\%$ capacity to the others. Suppose we let $5$ nodes have an answering power of $0.8$ and the other $5$ nodes with answering power of $0.1$. Running SLIC on this network, one might expect the nodes to form two cliques of five nodes: one for high answering power nodes and one for low answering power nodes. However, that is not true.

Figure 8 illustrates how the weights change for a node with high answering power over time. The x-axis gives the round number. The y-axis gives the weights. There are 9 curves, one for each link to the other nodes. Initially, the weights for links to the high answering power nodes increase while the weights for the low answering power nodes decrease dramatically. However, the weights for low answering power nodes do not all go to $0$; several of them stabilizes around $0.1$. The reason for this stability is because some low answering power nodes decided to process all queries from a high power node, thus ensuring its corresponding weight does not diminish to $0$.

An interesting question, then, is what kind of network results from running SLIC on this 10 nodes complete graph. Figure 9(a) shows the final network with weights for each edge. The edges not shown have weight $0$. In this figure, the high answering power nodes are colored black and low answering power nodes white. The style of the lines and arrows indicate different weights. To help decipher the data in the figure, solid lines have higher weights than dashed lines which have higher weights than dotted lines. Also for the same style of lines, a filled arrow indicated higher weight than a hollow arrow.

Obviously the high power nodes do prefer each other more. Fortunately, they still leak enough capacity to prevent the network from being disconnected. Notice the asymmetry in terms of the weights between a high power and a low power node. These asymmetric links also prevent the low answering power nodes from forming a clique of their own.

We also show what happens to the final network if every node uses $\rho = 0.4$, that is doubling the number of queries they each generate, in Figure 9(b). The network becomes less connected as nodes tend to pair up because the lack of capacity in the system. If we further increase $\rho$ for each node, the network will eventually become disconnected. As a side note, the precise configuration with weights is not entirely deterministic because the answering power of nodes introduce randomness. However, the general shape of the final configuration is similar.

From the two examples, we see that SLIC's greedy approach of adjusting weights do indeed capture a node's individual preference of their neighbors. What is unclear is how these locally determined weights interact on a global scale. In next section, we will show that each node's selfish decisions do indeed lead to a *good* incentive structure for the system as a whole where nodes are encouraged to share more data, give more capacity to other nodes, and establish more links to increase the network connectivity.

# 3 Incentive Structure

When running SLIC, a node will receive better service if its neighbors give it a high weight. To influence its neighbors decisions, a node has three options:

- Increase answering power. By sharing more data, a node can become more attractive.
- Increase the number of edges (or connectivity). By having more edges, a neighbor's queries can be forwarded to more nodes, which leads to more hits for neighbors' queries.
- Increase the amount of capacity used to service neighbors' queries. By giving more capacity, a node

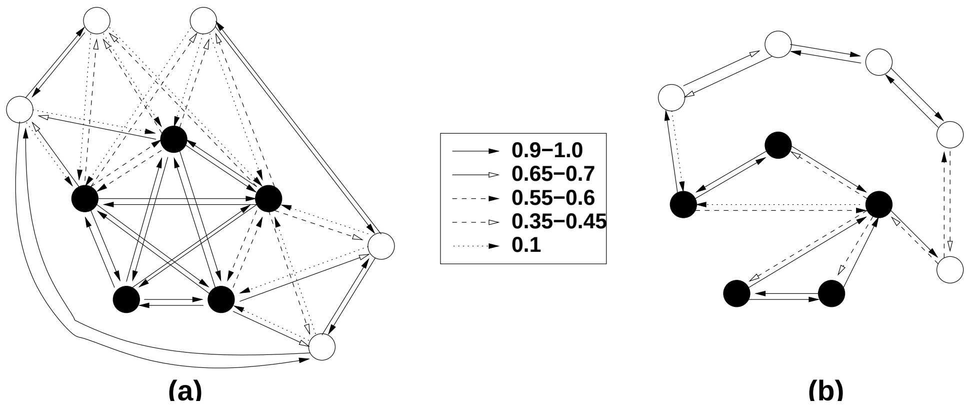

**Figure 9: Final network for running SLIC on complete graph of $10$ nodes with (a) $\rho = 0.2$ and (b) $\rho = 0.4$**

can forward more queries to reach distant parts of the network.

For the purpose of creating a good incentive structure, we also want SLIC to reduce a node's service if it does not provide a reasonable amount of resource in any of the above three categories. To assess the effectiveness of SLIC in establishing an incentive structure, we consider two utility functions:

1. $AvgHits_i(u)$: The average of number of hits per query generated by node $u$ in round $i$.
2. $TotalHits_i(u)$: The total number of hits for all queries generated by node $u$ in round $i$.

For the most part, both utility functions behave similarly, thus we will illustrate that SLIC has a good incentive structure by using $AvgHits_i(u)$. We will also highlight scenarios where $TotalHits_i(u)$ is a more appropriate utility function. With these two utility functions in mind, we will now demonstrate via simulation that SLIC rewards nodes that provide more data, dedicate more capacity for neighbors' queries, and establish more connections. Our result will also verify that SLIC ostracized nodes who do not play fair.

## 3.1 Answering Power

To assess the impact of varying answering power, we conducted simulations using $10$ randomly generated graphs of $250$ nodes where average node degree is $5$. (We also ran experiments with $250$ nodes power law topologies and larger graphs. The results show similar trends, but are not shown due to space limitations.) We first ran a baseline experiment where all nodes have an answering power of $0.4$, i.e., each node has a $40\%$ chance of having a hit for a query. For each node $u$, $\rho_u = 0.1$, or dedicating $90\%$ capacity for servicing neighbors' queries.

After collecting the baseline data, we then made one of the $250$ nodes a *probe node*. For this probe node, we varied its answering power from $0.1$ to $0.9$. Since graph structure and the location of the probe node also influence a node's quality of service, we ran multiple experiments with different graphs and probe nodes. With these different data points on different graphs, simply comparing the utility function $AvgHits_i(u)$ or $TotalHits_i(u)$ does not make sense. Instead, we compare the relative improvement or reduction in the the probe node's utility against the baseline data point. In particular, we compute the ratio of the utility of the probe node divided by the baseline utility when the probe node also had an answering power of $0.4$. Thus an improvement ratio of greater than 1 implies the node has received better service. Similarly, a ratio of less than 1 means diminished service. For this experiment, both $AvgHits_i(u)$ and $TotalHits_i(u)$ have the same behave, so we will only show the result for $AvgHits_i(u)$.

Figure 10 gives the result of our simulation with the associated confidence intervals. The x-axis shows different

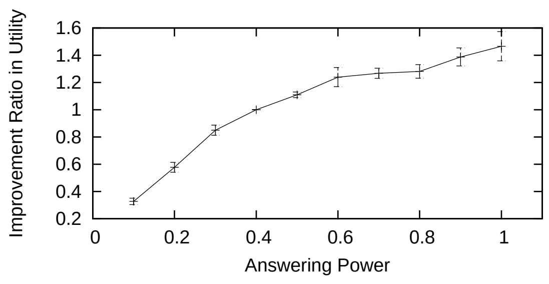

**Figure 10: Utility versus varying answering power.**

answering power for the probe node. The y-axis shows the average improvement ratio across different runs. As expected, the number of hits decreases almost linearly to $0$ if a node has a smaller answering power than the rest of the network. On the other hand, providing more answering power than the rest of the network does increase a node's utility, though less dramatically. The data point for the answering power $0.4$ does not have a confidence interval because it correspond to the baseline experiment where all the improvement ratios are 1. From this simulation result, we can conclude that a free-loader who shares much less data than an average user will have difficulty in obtaining quality service.

## 3.2 Connectivity

The number of links a node has directly influences the node's quality of service. Intuitively, if a node $u$ has many links, then its queries are serviced by more nodes. Moreover, when $u$ forwards one of its neighbor's query, it will also reach many nodes; thus the neighbors of $u$ will also give a high weight to $u$ as well, which in turns leads to better service for $u$. To quantify this intuition, we examine the utility of the nodes as a function of the node degree (connections). We again used 10 randomly generated graphs of $250$ nodes.

Figure 11 shows the result of the experiment with confidence intervals. On the x-axis is the node degree (i.e., number of connections). The y-axis shows the raw $AvgHits_i(u)$ utility value. (The utility function $TotalHits_i(u)$ behaves similarly.) Clearly, more connections result in much better service. However, note that the confidence intervals do have significant overlaps for

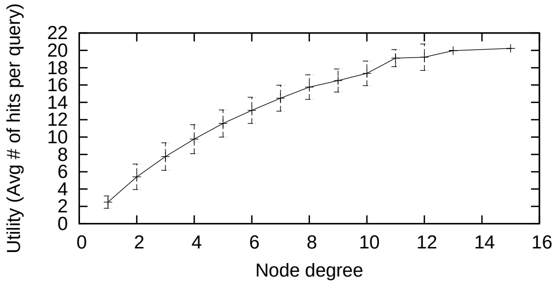

**Figure 11: Utility versus varying degree.**

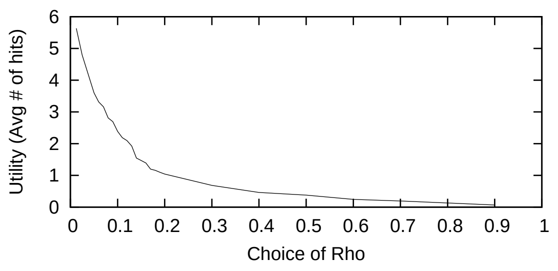

**Figure 12: Utility of a node with varying $\rho$.**

nodes with degree difference of 1. Thus to "guarantee" a significant improvement in utility, a node must increase its number of links by at least $3$ or $4$. Notice that the utility does not increase linearly with the node degree. The main cause is that eventually a node will run out of spare capacity to keep all of its neighbors happy. The data points for node degrees $13$ and $15$ do not have confidence intervals because there was only one node of that specific degree.

## 3.3 Spare Capacity

The parameter $\rho$ determines how many new queries are generated by a node each round. It also determines how much spare capacity is given to the neighbors. To see how this choice of balancing between injecting new queries and providing capacity to the neighbors affect utility, we first set $\rho = 0.1$ for all nodes. We then picked one probe node and varied its $\rho$ value.

Figure 12 shows the simulation result. The x-axis is the $\rho$ setting for the probe node. The y-axis shows the $AvgHits_i(u)$ utility. As one would hope, the utility drops exponentially as the node increases its $\rho$ to pump more queries into the system. However, this figure does not tell

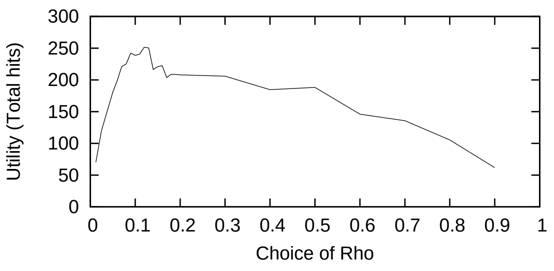

**Figure 13: Total hits with varying $\rho$.**

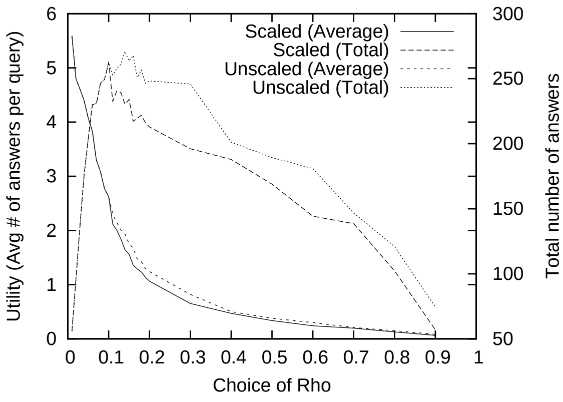

**Figure 14: Effect of using excess scaling**

the whole story. Figure 13 shows the same data except the y-axis now gives the $TotalHits_i(u)$ utility. We notice that the probe node can actually get more hits by pumping in a little bit more queries than the rest of the system (e.g., $\rho = 0.15$). It is possible, *though may not be desirable*, to prevent this phenomenon by scaling back a link's weight if the neighboring node is generating too many queries.

To prevent this loophole, we previously introduced the excess scaling modification *compute_weight_scaled* where a node penalizes its neighbor for generating too many queries. Figure 14 shows the result of applying the excess scaling for both utility functions. The figure has both the scaled and the unscaled data for comparison, hence four curves. We note that the $AvgHits_i(u)$ utility still drops off exponentially as the probe node generates more queries. With excess scaling, we have also prevented the probe node from getting more hits by generating more queries than the system norm of $\rho = 0.1$. On a cautionary note, in order for excess scaling to be effective, a node must have several neighbors to choose from. If a node $u$ only has one neighbor that is generating excess queries, then nothing will work except for node $u$ to disconnect and reconnect elsewhere.

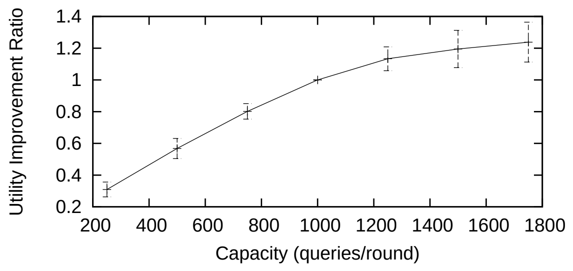

**Figure 15: Effect of processing capacity**

## 3.4 More Overall Capacity

A node may also obtain more spare capacity by simply allocating or buying more processing capacity. In this experiment, we varied the total processing capacity for a single probe node from $250$ to $1750$ queries per round, while all other nodes have a capacity of $1000$ queries per round. When changing the processing capacity, the probe node always uses $\rho = 0.1$ for generating new queries. Thus for larger capacity, the probe node is also generating more queries. The result is shown in Figure 15. The x-axis shows the processing capacity of the probe node. The y-axis shows the improvement ratio in terms of total number of hits, compared to the baseline of $1000$ queries per round. Not surprisingly, the improvement drops linearly when a node have less capacity than the rest. On the other hand, the improvement due to having more capacity flattens out. Performance flattens out because, even though neighboring nodes are preferring the probe node more and more, they still have the same amount of capacity for forwarding the probe node's queries as before. Therefore, when their capacity are exhausted, no more improvement is possible regardless how much extra capacity the probe node has.

## 3.5 Remarks

In this section we have demonstrated that using SLIC results in an incentive structure that encourages nodes to share more data, provide more capacity, and establish more connections. We have performed controlled experiments where we only varied one parameter at a time to quantify the effects. When multiple parameters are changing, the effects are not cumulative. For instance, adding new connections is likely to be more effective in terms of increasing utility than sharing more data because the benefit of reaching a new of group of nodes outweighs the benefit of a single node providing a little more data. For this precise reason, a free-loader who shares very little data can still thrive in a system running SLIC if it is willing to provide bandwidth that facilitate other nodes of reaching each other. We believe this scenario is actually desirable because the free-loader is not truly "free-loading" since it is providing valuable service to the system as a whole.

## 4 Dynamic Environment

The previous section has shown that SLIC can setup a good incentive structure under a static environment where the overlay network does not change. In order for SLIC to be usable in practice, it must also be able to adapt quickly to dynamic overlay network changes. There are many issues involved in dealing with dynamism. For example,

- When a new node joins, which existing nodes should it connect to?
- When a node receives a connection request, should it accept the connection unconditionally?
- When a new overlay link is created, what should the initial SLIC weight be?
- Should an overlay link with "very" low SLIC weight be dropped?
- Should an existing node with low utility attempt to create new overlay links to improve its utility?

In this section, we study two of the above five questions: initial weight for a new overlay links and allowing a node to create new links if it is unsatisfied.

### 4.1 Initial weight for new links

When a new link is created, a key design decision is how to initialize the weight for this link. We could initialize the weight to 1; however, this would allow free-loaders to drain resources from the system by reconnecting. We could also initialize the weight to a small value, which unfortunately creates a high barrier of entry for new nodes. Here we propose a simple solution and a couple of variations for initializing the weight of edge $(u, v)$ where $u$ is an old node and $v$ is a new node.

1. *Average*: Initialize $W_i(u, v)$ to be the average of the weights maintained by node $u$. In other words, if node $u$ has $d$ neighbors, then the new node $v$ is given $\frac{1}{d}$ of the spare capacity.
2. *Average_Inverse*: Initialize $W_i(u, v)$ to be the average weight multiplied by $\frac{1}{AvgHits_i(u)}$.
3. *Average_Exponential*: Initialize $W(u, v)$ to be the average weight multiplied by $e^{-AvgHits_i(u)}$

The *Average* scheme is fair in that it does not bias against a new connection, though it is susceptible to free-loaders. *Average_Inverse* and *Average_Exponential* address this concern by noting that if a node is already happy with its current utility, then there is little need to take a big risk in accepting a new connection. On the other hand, if a node is unhappy, then it might as well try its luck with a new node. Thus both variations adjusts the new weight by a function of the current utility with different aggressiveness.

To see how these variations of the average scheme perform, we ran simulations on our $250$ nodes random graphs where each node has an answering power of $0.4$ and $\rho = 0.1$. In these dynamic experiments, we first remove a node and its associated edges from the graph at the beginning and let the simulation run until it stabilizes. We then add the removed node and edges back into the network and continue the simulation to see how quickly SLIC responds. We also vary whether the node behaves normally or maliciously when it rejoins the network. For malicious behavior, we consider two cases: rejoining with low answering power and rejoining with high $\rho$ value (i.e., less spare capacity).

For brevity, we only show the results for one of our simulations where a node of degree 4 joined the network late and with different behaviors. (Although the exact numbers vary with different simulation setups, the general trends are identical to the results presented here.)

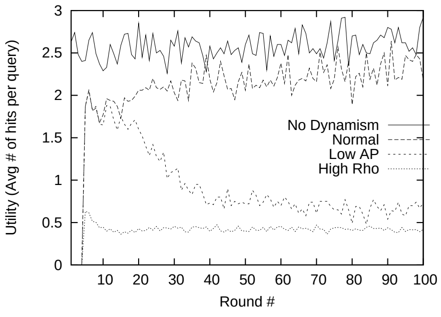

**Figure 16: Newly joined nodes**

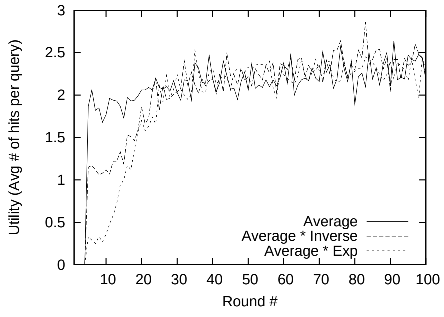

**Figure 17: A new normal node.**

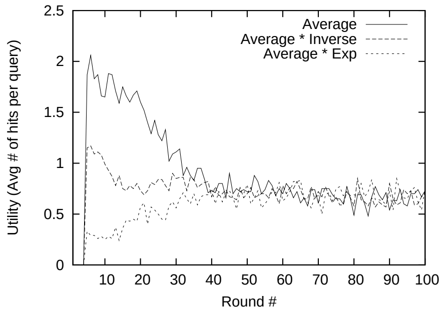

**Figure 18: A new low AP node**

Figure 16 shows the outcome of our simulation where the weight for the new edge is initialized to be the average weight. The x-axis shows the number of rounds since this node of degree 4 joined. The y-axis shows its $AvgHits_i(u)$ utility. For reference to the static environment, the top curve shows the utility of the node if it was part of the network since the beginning. The second curve shows that the utility of the node rapidly approaches the static case when it joined late and behaved normally. The third and forth curves illustrate that SLIC will respond to bad behavior quickly, even if the bad node joined late and was initially given a reasonable amount of spare capacity by its neighbors.

Of course, by using the two variations of average, we can reduce the resource drained by a malicious node at the expense of asking a good node to prove itself for a longer period. Figures 17 and 18 demonstrate this trade-off. Figure 17 shows that if the new node is behaving normally, then scaling down the new weight by either the inverse or an exponential will cause a delay of about $25$ rounds before the node reaches its proper utility level. At the same time, Figure 18 shows that a malicious node is not able to take advantage of the system as its utility settled down quickly without significantly exceeding its true level. Although *Average_Exponential* does perform quite well in our experiments, we believe *Average_Inverse* is more appropriate because for large networks where utility value is high, *Average_Exponential* maybe too aggressive in discriminating against new nodes.

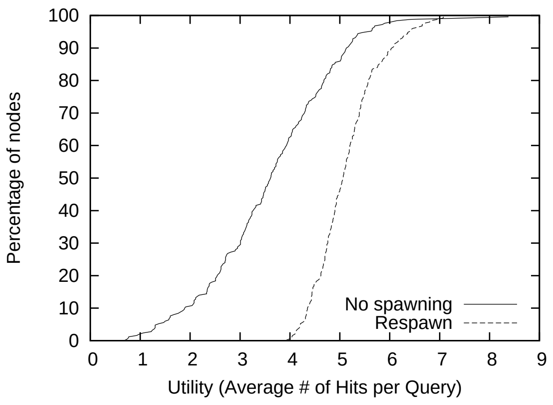

**Figure 19: CDF of $AvgHits_i(u)$ utility for the infinite growth scenario.**

## 4.2 Respawn Overlay Links

One fundamental principle in SLIC is to allow nodes to act selfishly and greedily to increase their own utility. Thus it is also natural for a node to create new overlay links if its own utility is too low. We now examine two variations of an unsatisfied node trying to establish new links. The first variation, named *Infinite Growth*, allows a node to add as many links as it wants. The second variation, named *Respawn Links*, restricts a node to maintain the same node degree, i.e., when creating a new link, it must drop an existing one. This second case is perhaps more realistic in that each node has a certain “budget” in the number links that they can support.

For the infinite growth variation, we ran a simulation on a $250$ node random graph where a node $u$ with $AvgHits_i(u) < 4$ will periodically try to create a new

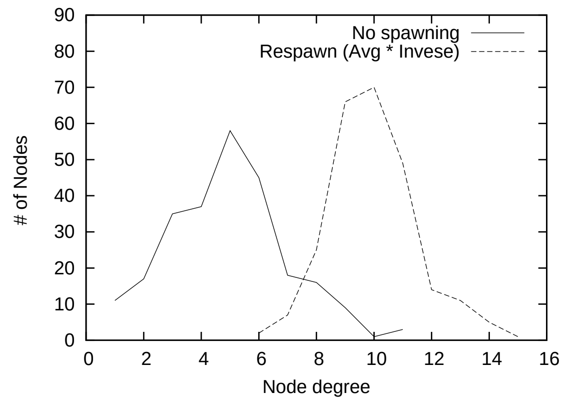

**Figure 20: Degree histogram for the infinite growth scenario.**

link to another node, chosen at random. The initial weight on the new link is determined by the $Average\_Inverse$ policy. For this run, all the nodes have answering power $0.4$ and $\rho = 0.1$. Figures 19 and 20 give the result of the simulation. Figure 19 shows the cumulative distribution plot of the $AvgHits_i(u)$ utility for all nodes when infinite growth was allowed and when it was not. As we can see clearly, with infinite growth, almost all nodes have $AvgHits_(u)$ utility of more than 4, which is not surprising. The price to achieve this level of utility is that every node has 5 more connections than before. This is evident from Figure 20 when we plot the histogram of nodes with a certain degree. We see the histogram shifted by about 5.

For the respawn links variation, we have to be careful to preserve the node degree. We use a four-way swap mechanism as illustrated in Figure 21. Suppose node $A$ is unsatisfied, and wants to create a new link to node $B$. Node $A$ must break an existing link to one of its neighbor, say node $C$. Similarly, node $B$ must also break an existing link, say to node $D$, to accept the new connection from $A$. We then create the link $(A, B)$. We also pair up $C$ and $D$ with a new link since they both have lost a connection. This swap clearly preserves the node degree at each node.

To evaluate SLIC under respawning, it does not make sense for all nodes to be identical because we are just replacing one graph of a given degree sequence with another of the same degree sequence. Thus there will not be significant change in terms of utility at all. Instead, we use three types of nodes: (1) normal nodes with answering power of $0.4$ and $\rho = 0.1$, (2) lowAP nodes with answering power of $0.1$ and $\rho = 0.1$, and (3) highRho nodes with answering power $0.4$ and $\rho = 0.5$. For our simulations, we used $250$ nodes random graphs with equal number of nodes for each type. When a node breaks a link, it chooses the link with the lowest weight.

We conducted simulations on different initial graphs with respawning where a node chooses another node at random when it wants to exchange neighbors. We also force a node to accept a new link if asked. We use *Average_Inverse* for the new edge weight. For each run, we first ran a simulation for $5000$ rounds. We then for each node computed how its $AvgHits_i(u)$ utility has improved or deteriorated as a ratio against the baseline comparison when there is no respawning. For brevity, we present the result from one run in Figure 22. (Other runs produced similar results.) We plot three cumulative distribution plots, one for each type of nodes (normal, lowAP, and highRho). The x-axis shows the improvement ratio (e.g., bigger than 1 means better utility). The y-axis shows the percentage of nodes. Notice that for a normal node that is behaving properly, $80\%$ of them have improved utility (ratio greater than 1) after respawning. In contrast, $70\%$ of lowAP nodes and $80\%$ of highRho nodes experienced reduction in utility (ratio less than 1). Also notice that there is a significant gap between the normal nodes and the malicious nodes in their improvments. Therefore we can confidently conclude that when allowing nodes to reconnect, good nodes will derive great benefits while bad nodes cannot take significant advantages of the system.

As previously discussed, nodes that give less capacity to the system are penalized more severely than nodes sharing less data. This intuition is verified in Figure 22 as the lowAP nodes experience less deterioration in utility than highRho nodes. Curiously, one might expect that nodes with high degree would achieve better improvement ratio. This intuition is incorrect as can be seen in the scatter plot of node degree versus improvement ratio in Figure 23. There is no trend or clustering to draw any correlation between node degree and improvement.

## 5 Generalizing SLIC

The greedy SLIC approach is quite general: a node provides service, and in return receives service from others.

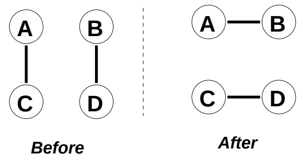

**Figure 21: Four-way swap of respawning overlay links**

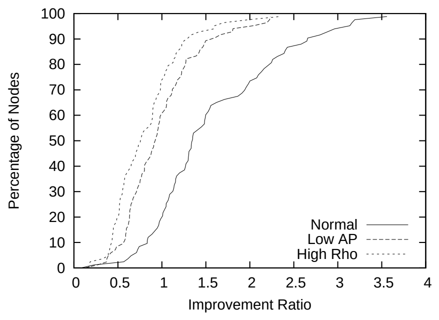

**Figure 22: CDF of utility improvement**

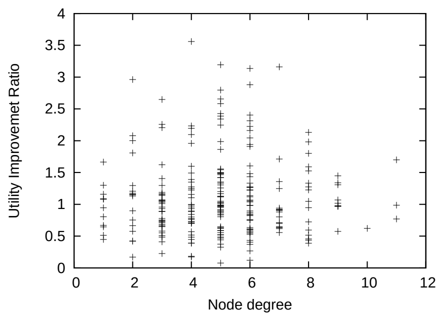

**Figure 23: Node deg. vs. improvement**

The amount of service it provides a neighbor is related to the service a node receives, and this creates the right incentives. The paper has dealt with only one type of service, answering queries, but the idea can be used for other services, e.g., downloading files, indexing content of nearby nodes, doing computations, etc.

If a node provides a variety of services (e.g., query answerng and file download) one can either: (1) aggregate services (rendered or received) into a single metric, so a node can tell the overall level of service received from others, or (2) run SLIC in parallel, where each class of service is handled separately. For example, in aggregrating query answering and file download into a single metric, a successful file download can equate to, say, 50 search hits.

## 6 Related Work

Similar to SLIC's retaliation-based approach, BitTorrent [4], a P2P client for sharing a single file, uses a similar idea for controlling download/upload rates between participating clients. Our greedy approach is also similar to the game theoretic approach of [9] where Lai et. al. analyze the impact of a type of tit-for-tat strategy. In contrast to our local approach, reputation-based and trust-based systems enforce a global structure. For example, the EigenTrust algorithm [8] "collects" all pair-wise preference values between peers and computes the eigenvector as the global reputation. Alternatively, Cornelli et. al. in [5] propose a voting scheme where a peer solicits "votes of confidence" when deciding between which peers to download data from. One can also translate the notion of trust into hard currency by using payment-based ideas like Mojo Nation [11] where peers earn credits for providing service.

## 7 Concluding Remarks

We have demonstrated that our simple Selfish Link-based InCentive (SLIC) creates a desirable incentive structure for unstructured P2P file sharing systems where nodes in exchange for better service are encouraged to share more data, give more capacity to neighboring nodes' queries, and add new overlay links. Moreover, if nodes dynamically adjust their links to strive for better service, SLIC responds quickly and fairly for nodes that are playing by the rule to improve their quality while not letting malicious nodes take advantage of the situation.

## References

[1] K. Aberer and Z. Despotovic. Managing trust in a peer-to-peer information system. In $10^{th}$ *International Conference on Information and Knowledge Management (CIKM)*, 2001.

[2] E. Adar and B. Huberman. Free riding on gnutella. *First Monday*, 5(10), October 2000.

[3] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker. Making gnutella-like p2p systems scalable. In *Proceedings of ACM SIGCOMM*, 2003.

[4] B. Cohen. Incentives build robustness in bittorrent. In *Workshop on Economics of Peer-to-Peer Systems*, 2003.

[5] F. Cornelli, E. Damiani, S. D. C. di Vimercati, S. Paraboschi, and P. Samarati. Choosing reputable servents in a p2p network. In $11^{th}$ *International WWW Conference*, 2002.

[6] R. Dingledine, M. J. Freedman, and D. Molnar. The free haven project: Distributed anonymous storage service. In *Workshop on Design Issues in Anonymity and Unobservability*, 2000.

[7] Gnutella. http://gnutella.wego.com.

[8] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina. The eigentrust algorithm for reputation management in p2p networks. In $12^{th}$ *International WWW Conference*, 2003.

[9] K. Lai, M. Feldman, I. Stoica, and J. Chuang. Incentives for cooperation in peer-to-peer networks. In *Workshop on Economics of Peer-to-Peer Systems*, 2003.

[10] S. Lee, R. Sherwood, and B. Bhattacharjee. Cooperative peer groups in nice. In *IEEE INFOCOM*, 2003.

[11] MojoNation. http://www.mojonation.com/.

[12] Q. Sun, N. Daswani, and H. Garcia-Molina. Maximizing remote work in flooding-based peer-to-peer systems. In $17^{th}$ *International Symposium on Distributed Computing*, 2003.

[13] B. Yang and H. Garcia-Molina. Ppay: Micropayments for peer-to-peer systems. In *10th ACM Conference on Computer and Communications Security (CCS)*, 2003.