Data Transfer Scheduling for P2P Storage

Laszlo Toka, Matteo Dell'Amico, Pietro Michiardi
{laszlo.toka, matteo.dell-amico, pietro.michiardi}@eurecom.fr
Eurecom, Sophia-Antipolis, France

***Abstract*—In Peer-to-Peer storage and backup applications, large amounts of data have to be transferred between nodes. In general, recipient of data transfers are not chosen randomly from the whole set of nodes in the Peer-to-Peer networks, but they are chosen according to peer selection rules imposing several criteria, such as resource contributions, position in DHTs, or trust between nodes. Imposing too stringent restrictions on the choice of nodes that are eligible to receive data can have a negative impact on the amount of time needed to complete data transfer, and scheduling choices influence this result as well. We formalize the problem of data transfer scheduling, and devise means for calculating (knowing *a posteriori* the availability patterns of nodes) optimal scheduling choices; we then propose and evaluate realistic scheduling policies, and evaluate their overheads in transfer times with respect to the optimal. We show that allowing even a small flexibility in choosing nodes after the peer selection step results in large improvements on time to complete transfers, and that even simple informed scheduling policies can significantly reduce transfer time overhead.**

I. Introduction

One of the most appealing characteristics of Peer-to-Peer storage and backup applications is that very large amounts of data can be stored at low costs, adopting excess free space in internal hard drives and/or removable devices; these applications therefore require moving large amounts of data between nodes. The current state of technology implies that amounts of data that can comfortably fit on today's disk drives require a long time to be transferred: for example, on an ADSL line having a standard 1Mbps upload speed, 10 GB of data need almost a day of continuous upload. In addition, the unreliability of peers as storage nodes require data to be sent with redundancy and the fact that many nodes only spend a few hours online can further extend the amount of time needed to complete data transfers. Moreover, time to transfer has a strong impact over data durability: as long as data is not uploaded with redundancy to nodes, it risks being lost in the case of a local disk crash: in realistic scenarios, the most likely cause for data loss can be the simple fact that nodes experience a disk crash before completing the data upload [1].

The length of data transfers and their impact on data durability motivates the usage strategies that shorten them. In this work, we focus on how to deal with situations where the amount of data to transfer is large, and the amount of resources invested in these transfers requires keeping track of the data stored on nodes that are transiently offline: when they will reconnect, data stored at them will be available again.

Storage applications generally impose restrictions on data placement, often making only a given set of peers eligible for storing some particular chunks of data. We call *peer selection* the process that enforces these restrictions and outputs a list of peers that are eligible for storing data, which we will term *peer set*. Each node in a peer set is eligible to store a limited amount of data, depending both on storage capabilities and application requirements (e.g., when using erasure coding, storing too much data on the same node can put characteristics such as data availability or durability at stake).

These restrictions govern the design of Peer-to-Peer applications: for example, data can be placed on particular nodes of a distributed hash table in order to facilitate locating it [2], [3]; restrictions can be imposed in order to store data with particular storage or uptime characteristics [4], also to create incentives to cooperation [5]; in systems such as FriendStore [6] or Safebook [7] data is only placed on machines owned by friends of the data owner, in order to leverage on trust relationships between users. In addition to these reasons, smaller peer sets reduce the "book-keeping" costs due to monitoring nodes holding data. Unfortunately, these restrictions can determine performance issues when coupled with the phenomenon of *churn*, *i.e.* the frequent and unpredictable pattern of connections and disconnections of peers [8]. If the restrictions imposed by peer selection are too rigid – resulting in small peer sets – there can be high overheads in terms of time needed to complete transfers: for example, if none of the eligible nodes are online when transfers occur, the data owner will have to wait until one of them returns online. If the peer set is bigger, such a situation is less likely to arise.

In this work, we propose a model for data transfers, accounting for uploads from a single source node towards multiple destinations, as well as downloads from multiple sources, adopting erasure coding. We consider the time to transfer (TTT) as our main metric, and we investigate how it is related to the size of peer set, maximum data uploaded to each remote node, and the *scheduling policy* adopted for uploads in the presence of churn. Indeed, scheduling choices (*i.e.*, deciding how to allocate upload bandwidth towards remote nodes) impact on TTT; for example, if the currently online nodes in the peer set have already received their maximum amount of data, no transfer is possible; other scheduling choices could have resulted in data transfers not stalling.

When nodes are homogeneous, in terms of bandwidth and connectivity behavior, scheduling choices are not significant. Conversely, we show that the peer heterogeneity observed in real applications makes scheduling matter, since informed data transfer policies can proactively avoid a situation where uploads stall. In related work, peer connectivity patterns are usually taken into account using simple mathematical model-

ing, often using memory-less processes where the probability that a node disconnects is the same for each node and at each time. In this case, the scheduling problem becomes trivial: since all nodes have the same behavior, choosing one or the other is indifferent. However, since it is guided by human behavior, churn is not a completely random process: node availability exhibits regularities such as diurnal and weekly patterns, and different behavior between users [9]. Trying to create more complex churn models that attempt to describe all the particularities and regularities of user behavior would be prohibitive, given the inherent complexities of human behavior. We instead take a different approach, using availability traces *as inputs* of the scheduling problem: this approach guarantees that the evaluation of scheduling policies will not be affected by artifacts or simplifications due to the model. In Section II, we formally define the scheduling problem, and define scheduling choices as a function of past traces.

We define optimal scheduling based on *a posteriori* knowledge of the whole node traces: this will provide us a useful baseline against which we compare feasible online scheduling choices, based only on knowledge of node uptime up to when a scheduling choice is made. While the problem may appear difficult to solve in polynomial time, in Section III we show that the optimal scheduling can indeed be calculated efficiently. First, we model the case when no congestion happens: only a single node is uploading its data. We show that a maximal flow formulation can be used to solve our problem in polynomial time. In Section IV, we show – after relaxing some constraints – that the problem of completing several concurrent transfers can be seen as a linear programming problem which is solvable in polynomial time.

Knowing future node uptime in advance is obviously impossible. In Section V, we therefore propose and discuss several policies where scheduling decisions depend on past traces. In Section VI, we evaluate experimentally the performance of these policies, using real traces as simulation inputs. Based on these results, we conclude that the most important factor influencing the time needed to complete uploads is the number of nodes present in peer sets: as this value grows, the time to complete transfers decreases rapidly. This is an important message to application designers: allowing a small degree of flexibility for the choice of nodes to adopt in the overlay pays off significantly. In addition, we discovered that congestion has only a moderate impact on the amount of time needed to complete data transfers, imposing small penalties with respect to cases where a single node is sending data, mainly due to asymmetric up- and downlinks. Finally, we show that our proposed scheduling policies help significantly reduce time to transfer, with a decrease in the overhead due to non-optimal scheduling by a factor of around 40% in our experiments.

II. The Scheduling Problem

The way scheduling is chosen can obviously impact the amount of time needed to complete a data transfer. Consider the availability traces in Figure 1, where the data owner has a unitary upload speed per timeslot, and has to upload one

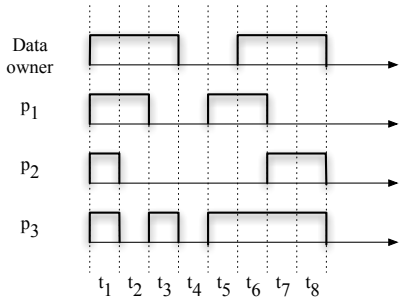

Figure 1: Example availability traces.

data unit per remote peer – one each to p_1, p_2 and p_3. With an optimal schedule the owner would send a unit to p_2 in the first timeslot, then one to p_1 in timeslot t_2, then one to p_3 concluding the transfer in timeslot t_3. Conversely, if data is sent to p_3 during the first timeslot, in the second timeslot p_1 is the only possible choice. The transfer will have to stall until p_2 comes back online in timeslot t_7.

In the rest of this Section, we formally define the scheduling problem as an optimization problem for transfers from a single data source towards nodes within a predefined peer set, having bandwidth and availability of nodes as inputs. We then show how the same formalization can also apply to downloads. In Section IV we extend the model to cater for a congested case, with several nodes performing uploads concurrently.

A. Problem Formulation

In our scenario, a data owner has a peer set of n nodes to which it needs to upload a data object of size o. We take traces as an input of our problem, encompassing T timeslots (starting from the beginning of the upload process) in which peer availability and bandwidth can change; within a single timeslot the network conditions remain stable. In our model, the timeslot duration is not constrained to be constant. We do not consider cases (such as network coding or simple replication) where data destined to a node can be obtained from data sent to other peers, and we therefore only consider data to be uploaded from the data owner. In most cases data will be divided in fragments and incomplete fragment transfers will be discarded; we consider the fragment size to be small enough to consider the amount of data lost in this way (*e.g.*, nodes disconnecting while receiving data) negligible.

We model the network congestion as limited by the upload/download bandwidth of peer access links, and we assume that bandwidth can vary between timeslots: we thus model u_t as the amount of data that the owner can upload within timeslot t, and $d_{i,t}$ as the amount of data that peer i can download in the same timeslot t. We express the fact that nodes are offline in a timeslot by setting the corresponding bandwidth to 0.

We impose an additional constraint m_i on the maximum data that can be uploaded to each peer i. This restriction can be due to both storage capabilities of nodes and to system design choices (for example, if the data owner is uploading the result of an erasure coding process, having too much data

Symbol	Meaning in the upload scenario	Meaning in the erasure-coded download scenario
n	peer set size	number of remote data holders
T	number of timeslots	number of timeslots
o	data object size	data object size
$d_{i,t}$	data peer i can download in timeslot t (0 if i is offline)	data peer i can upload in timeslot t (0 if offline)
u_t	data owner can upload in timeslot t (0 if owner is offline)	data owner can download in timeslot t (0 if offline)
m_i	maximum amount of data uploaded to peer i	data stored on peer i

Table I: Inputs of the scheduling problem.

on the same node could degrade data availability or durability).

With this problem formulation, we aim to optimize the time to transfer of a *single* node. Network congestion and storage limitations caused by the simultaneous activity of other nodes can be expressed by altering the values of $d_{i,t}$ and m_i, subtracting the resources allocated to other peers. In Section IV, we propose a model where the goal is to optimize the time to complete several concurrent transfers on the same network.

The notation is summarized in the left part of Table I.

Definition 1. The *ideal time to transfer* (idealTTT) is the minimum number of timeslots needed to upload the data object considering only the bandwidth of the data owner:

$$idealTTT = \min\left\{t \in 1\dots T : \sum_{i=1}^{t} u_i \geq o\right\}.$$

IdealTTT represents the time to transfer data when uploading to an ideal server, which is supposed to be always online and to have enough bandwidth to saturate the owner's uplink. The differences between idealTTT and time to transfer values observed for P2P systems are entirely due to the limits of remote nodes and to the inefficiency of scheduling policies.

Definition 2. A *schedule* S represents the amount of data sent to each node during each timeslot. For each peer i and timeslot t, we will denote $S(i,t)$ as the amount of data sent to peer i during timeslot t. During a timeslot t, a node can send data concurrently to several peers at once, reflecting the case where a node uploads data in parallel to several destinations. A schedule has to satisfy the following conditions.

1) Upload constraints are respected:

$$\forall t \in [1,T] : \sum_{i=1}^{n} S(i,t) \leq u_t. \tag{1}$$

2) Download constraints are respected:

$$\forall i \in [1,n], t \in [1,T] : S(i,t) \leq d_{i,t}. \tag{2}$$

3) Storage constraints are respected:

$$\forall i \in [1,n] : \sum_{t=1}^{T} S(i,t) \leq m_i. \tag{3}$$

We denote the set of all schedules as $\mathcal{S}$.

Definition 3. A schedule S is *complete* if at least a total amount o of data has been transmitted:

$$\sum_{i=1}^{n}\sum_{t=1}^{T} S(i,t) \geq o. \tag{4}$$

We denote the set of all complete schedules as $\mathcal{CS}$.

Definition 4. The *time to transfer* (TTT) of a schedule S is its completion time, *i.e.* the last timeslot in which the data owner uploads data:

$$TTT(S) = \max\left\{t \in [1,T] : \sum_{i=1}^{n} S(i,t) > 0\right\}.$$

The goal of a scheduling policy is to obtain the shortest possible TTT. In Section III we show how to evaluate *a posteriori* an optimal scheduling policy given the traces, and in Section V we discuss how to choose a scheduling policy.

B. Extension to Downloads

Let us consider the case of a single node downloading a piece of content composed of data fragments stored on remote peers. When these fragments are encoded with erasure coding techniques, there is a value k such that any k fragments are sufficient to recover the original data.[1]

Scheduling can be covered with the very same formalization discussed above, inverting the direction of the data flows: Table I shows how notations map to their meaning in this situation. Also in this case there is a data amount o to transport in total; the n nodes in the peer set are the remote data holders; the constraints of the problem take different meanings, but they have the exact same role with respect to our model. The values m_i take the role of the data amount that is remotely stored on each node – any amount of data between 0 and m_i can be transferred from i; the meaning of the $d_{i,t}$ and u_t values are now inverted, mapping now to respectively peer i's upload bandwidth and the owner's downlink. The equations of Definitions 2 and 3 still have to hold, requiring that constraints are respected for download (Equation 1), upload (Equation 2) and storage (Equation 3), and at least o bytes are transferred (Equation 4). The definitions of idealTTT and TTT still hold with the same meaning.

Given that both cases map to the same problem in the following we will refer, without loss of generality, only to the case of upload described before.

III. Optimal Scheduling

Once the whole traces are known, it is possible to compute the minimal possible time to transfer. Obviously, this knowledge is not available in real time, so this information cannot be

[1]Non-optimal coding techniques (*e.g.*, LT codes [10]) give probabilistic guarantees on the ability to decode data with $k' = k + \varepsilon$ fragments. They can be taken into account by choosing an ε big enough to give sufficient guarantees and using k' instead of k.

used to devise scheduling policies. Nevertheless, it is possible to compare *a posteriori* the optimal TTT with the results obtained using online scheduling policies in order to evaluate their degree of efficiency. Despite the fact that finding optimal scheduling may appear computationally very expensive at first sight, we devise an efficient polynomial solution based on a max-flow formulation.

The rationale and performance for scheduling policies are deeply intertwined with the characteristics of node churn. In Section V we discuss the choice of scheduling policy, and in Section VI we experimentally evaluate scheduling policies, and their overhead with respect to optimal scheduling, on real availability traces.

Definition 5. The *optimal time to transfer* (optTTT) is the minimal TTT within the set of all complete schedules $\mathcal{CS}$:

$$optTTT = \min\left\{TTT(S) : S \in \mathcal{CS}\right\}. \tag{5}$$

We use optimal TTT as a baseline to compute the overhead in time-to-transfer for a given scheduling policy.

Definition 6. The *scheduling overhead* for a schedule S is the relative increase in TTT due to a non-optimal scheduling:

$$\frac{TTT - optTTT}{optTTT}.$$

We compute optimal scheduling by solving several instances of the related problem: "how much data can be transferred within the first $\bar{t}$ timeslots"? We will use the following Proposition to relate the two problems.

Proposition 7. Let $\mathcal{S}$ be the set of all schedules, and $F(\bar{t})$ be the maximum amount of data that can be uploaded not later than $\bar{t}$, that is:

$$F(\bar{t}) = \max\left\{\sum_{i=1}^{n}\sum_{t=1}^{\bar{t}} S(i,t) : S \in \mathcal{S} \wedge TTT(S) \leq \bar{t}\right\}; \tag{6}$$

Optimal TTT will be

$$optTTT = \min\left\{\bar{t} \in 1 \dots T : F(\bar{t}) \geq o\right\}. \tag{7}$$

Proof: Let $t_1 = optTTT$ and $t_2 = \min\left\{\bar{t} \in 1 \dots T : F(\bar{t}) \geq o\right\}$. We show that both $t_1 \geq t_2$ and $t_1 \leq t_2$ hold.

1) $t_1 \geq t_2$. By Equation 5, an $S_1 \in \mathcal{CS}$ exists such that $TTT(S_1) = t_1$ and, since $S_1 \in \mathcal{CS}$, by Equation 4 $\sum_{i=1}^{n}\sum_{t=1}^{T} S(i,t) \geq k$. The existence of S_1 implies that $F(t_1) \geq o$ (Equation 6) and therefore $t_2 \leq t_1$.
2) $t_1 \leq t_2$. By Equations 6 and 7, an S_2 exists such that $TTT(S_2) = t_2$ and $\sum_{i=1}^{n}\sum_{t=1}^{T} S(i,t) \geq o$. This directly implies that $t_1 = optTTT \leq t_2$.

■

The former Proposition allows us to find optTTT by computing different values of $F(t)$ and outputting the smallest value $\bar{t}$ such that $F(\bar{t}) \geq o$.

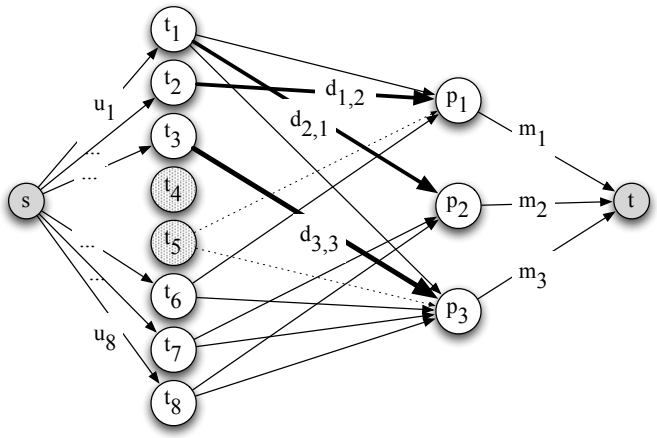

Figure 2: Flow network equivalent to the traces of Figure 1 on page 2.

A. Max-flow Formulation

Let us now focus on how to compute $F(t)$. This problem can now be encoded as a max-flow problem on a network built as follows. First, we create a complete bipartite directed graph $G' = (V', E')$ where $V' = \mathcal{T} \cup \mathcal{P}$ and $E' = \mathcal{T} \times \mathcal{P}$; the elements of $\mathcal{T} = \{t_i : i \in 1 \dots T\}$ represent time-slots, the elements of $\mathcal{P} = \{p_i : i \in 1 \dots n\}$ represent remote peers. Source s and sink t nodes are then added to the graph G' to create a flow network $G = (V, E)$. The source is connected to all the time-slots during which the data owner is online; all peers are connected to the sink.

The capacities on the edges are defined as follows: each edge from the source s to time-slot i has capacity u_i; each edge between time-slot t and peer i has capacity $d_{i,t}$; finally, each edge between peer i and the sink has capacity m_i.

Since we are interested in maximal flow, we can safely ignore (and remove from the graph) those edges with capacity 0 (corresponding to nodes that are offline). In Figure 2, we show the result of encoding the example of Figure 1.

We will show that each $s \to t$ network flow represent a schedule, and a maximal flow represents a schedule transferring the maximal data $F(t)$. In the example of Figure 2, the bold edges represent a solution to the maximal flow problem on the first 3 timeslot nodes where an amount of data $o = 3$ is uploaded, with a flow of 1 per edge.

A nonzero flow from a timeslot node to a peer node represents the data uploaded towards that node in the specific timeslot; parallel transfers happen when multiple outgoing edges from a single timeslot node have nonzero flow. The constraints that guarantee that the schedule is valid according to Definition 2 on the previous page are guaranteed by the edge labels: upload constraints (Equation 1) are guaranteed by edges from source to timeslot; download constraints by edges from timeslots to peers (Equation 2); storage constraints by edges from peers to the sink (Equation 3).

Algorithm 1 Algorithm for finding optTTT.

$l \leftarrow 1$; $r \leftarrow 1$
% We look for a r value with $D(r) \geq o$.
% In this cycle, maximum $\log_2 \bar{t}$ invocations to D.
while $D(r) < o$:
    $l \leftarrow r$; $r \leftarrow 2r$
% Now $l \leq \bar{t} \leq r$; we look for $\bar{t}$ via binary search.
% Again, maximum $\log_2 \bar{t}$ invocations to D.
while $l \neq r$:
    $t \leftarrow \left\lfloor \frac{l+r}{2} \right\rfloor$
    if $D(t) < o$:
        $l \leftarrow t$
    else:
        $r \leftarrow t$
return l

B. Computational Complexity

As guaranteed by Proposition 7 on the preceding page, optTTT can be obtained by finding the minimum value $\bar{t}$ such that $F(\bar{t}) \geq o$. The $\bar{t}$ value can be found by binary search, requiring $O\left(\log \bar{t}\right)$ calls to the routine computing F as in Algorithm 1; for a flow network with V nodes and E edges, the max-flow can be computed with time complexity $O\left(VE \log\left(\frac{V^2}{E}\right)\right)$ [11]. In our case, when we have n nodes and an optimal solution of $\bar{t}$ time-slots, V is $O(n+\bar{t})$ and E is $O(n\bar{t})$. The complexity of an instance of the max-flow algorithm is thus $O\left(n\bar{t}\left(n \log \frac{n}{\bar{t}} + \bar{t} \log \frac{\bar{t}}{n}\right)\right)$. Multiplying this by the $O\left(\log \bar{t}\right)$ times that the max-flow algorithm will need to be called, we obtain a computational complexity for the whole process of $O\left(n\bar{t} \log n \left(n \log \frac{n}{\bar{t}} + \bar{t} \log \frac{\bar{t}}{n}\right)\right)$.

IV. Model With Congestion

With the formalization of Section II, we defined the scheduling problem as an optimization problem concerning only the scheduling choices of a single data owner, and aimed to optimize its TTT. The network congestion due to other nodes competing for bandwidth at the same nodes could be expressed by varying the values of u_t and $d_{i,t}$, but the scheduling choices adopted by those nodes could not be changed in order to accommodate for a more efficient global behavior.

We now consider the case of a Peer-to-Peer application where nodes behave altruistically, performing their choices in order to optimize towards the common good of the community.[2] In this case, we consider the case where the n nodes in the network have data objects to back up, and we formalize the problem of minimizing a *global time to transfer*: completing as soon as possible *all* data transfers.

In Table II we enumerate the new inputs of the problem, generalizing the inputs of the single-node scenario of Table I on page 3. For the download scenario, the same different meanings would apply here. Each node i has a data object of size o_i to upload ($o_i = 0$ if node only i stores data without uploading anything); the upload speed can now vary per node, where $u_{i,t}$ is the amount of data that i can upload in timeslot t; the data that j can store for i is now expressed as $m_{i,j}$ ($m_{i,j} = 0$ if $i = j$ or j is not part of i's peer set and cannot therefore store i's data). In addition, the overall storage capacity of node i is now expressed as $\overline{m_i}$. We can now formulate the definition of a schedule and a complete schedule in this case.

Symbol	Meaning in the upload scenario
n	network size
T	number of timeslots
o_i	data object size for peer i
$d_{i,t}$	data peer i can download in timeslot t (0 if i is offline)
$u_{i,t}$	data peer i can upload in timeslot t (0 if i is offline)
$m_{i,j}$	maximum data uploaded by peer i on j (0 if $i = j$)
$\overline{m_i}$	storage capacity on node i

Table II: Inputs of the scheduling problem with congestion.

Definition 8. A *global schedule* G represents the amount of data sent between each pair of nodes during each timeslot. For each pair of nodes (i, j) and timeslot t, we denote the amount of data sent from i to j during timeslot t as $G(i, j, t)$. Also in this case, parallel transfers are supported by the model: a node can send data to several recipients at once, and a recipient can receive transfers from several origins concurrently. A global schedule has to satisfy the following conditions.

1) Upload constraints are respected:

$$\forall i \in [1,n], t \in [1,T] : \sum_{j=1}^{n} G(i,j,t) \leq u_{i,t}. \quad (8)$$

2) Download constraints are respected:

$$\forall j \in [1,n], t \in [1,T] : \sum_{i=1}^{n} G(i,j,t) \leq d_{i,t}. \quad (9)$$

3) Storage constraints are respected:

$$\forall i,j \in [1,n]^2 : \sum_{t=1}^{T} G(i,j,t) \leq m_{i,j}; \quad (10)$$

$$\forall j \in [1,n] : \sum_{i=1}^{n} \sum_{t=1}^{T} G(i,j,t) \leq \overline{m_j}. \quad (11)$$

Definition 9. A global schedule G is *complete* if each node i has trasmitted o_i data:

$$\forall i \in [1,n] : \sum_{j=1}^{n} \sum_{t=1}^{T} G(i,j,t) \geq o_i. \quad (12)$$

Definition 10. The *global time to transfer* (GTTT) of a global schedule G is

$$GTTT(G) = \max \left\{ t \in [1,T] : \sum_{i=1}^{n} \sum_{j=1}^{n} G(i,j,t) > 0 \right\}.$$

[2] This case applies for example to "managed Peer-to-Peer" settings in which peers are machines run and administered by the same entity, such as set-top-boxes used for Internet access.

A. Optimal Scheduling

We now delve in the way to compute optimal scheduling in this case. Analogously to what we did in Section III, we first formulate the problem of computing the amount of data $MAXD(\bar{t})$ that can be transmitted within the first $\bar{t}$ timeslots, and we use iteratively the solution of this problem to find the smallest $\bar{t}$ value that satisfies the condition of completing all transfers. We formalize the aforementioned problem in the following definition.

Definition 11. $MAXD(\bar{t})$ is the maximum amount of data that all nodes can transfer within timeslot $\bar{t}$. It is determined by the linear programming optimization problem

$$MAXD(\bar{t}) = \max \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{t=1}^{\bar{t}} G(i,j,t) \tag{13}$$

subject to the constraints of Equations 8, 9, 10, 11, and

$$\forall i \in [1,n] : \sum_{j=1}^{n} \sum_{t=1}^{T} G(i,j,t) \leq o_i \tag{14}$$

enforcing that no peer uploads more than its o_i-bytes.

It is worthwhile to discuss the characteristics of this formalization. The MAXD problem determines a global schedule with n^2T variables (*i.e.*, all the $G(i,j,t)$ values). Constraints 8, 9, 10, 11 and 14 express valid linear programming equations, as each one of them forces a linear combination of the values in G to be less than or equal to one of the scalar values of the inputs: respectively, the nT values of $u_{i,t}$ and $d_{i,t}$, the n^2 values of $m_{i,j}$, the n values of $\overline{m_i}$ and the n values of o_i. Overall, this results in a constraint matrix with $n\,(2T+n+2)$ equations and n^2T variables. Since it is a linear programming problem, MAXD can be solved in polynomial time [12], [13].

Definition 12. The *optimal global time to transfer* (optGTTT) is the first value $\bar{t}$ for which each node i can transfer its o_i bytes:

$$optGTTT = \min \left\{ t \in [1,T] : MAXD(t) \geq \sum_{i=1}^{n} o_i \right\}.$$

Analogously to optTTT with Algorithm 1 on the preceding page, optGTTT can be found by binary search with $O(\log \bar{t})$ invocations to the routine computing MAXD.

V. Scheduling Policies

As opposed to the optimal scheduling considered until now, we now move on to discuss strategies that can can actually be implemented, meaning that a scheduling decision applied at time t is only dependent on information that is available at time t. For convenience, we will use $a_{i,t}$ as a binary value assuming value 1 if peer i is online at timeslot t:

$$a_{i,t} = 1 \,\text{if}\, d_{i,t} \neq 0,\ 0 \,\text{otherwise}.$$

Each of the scheduling policies we introduce in this Section gives a priority value $v_i(t)$ to each node i at time t. The scheduling policy chooses to upload data to the available node in the peer set with the highest priority value. In case of ties, we break them by selecting nodes randomly. If the highest-priority node is unavailable or the upload speed of the data owner is not saturated, further nodes are selected by descending order of priority.

It is worthy to note that adopting different scheduling policies may result in different system properties (e.g., if uploads are prioritized towards low-availability nodes, data that is uploaded could have a lower availability). This issue is outside of the topic of this paper, since we consider that the required system properties should be ensured by proper policies within the peer set selection step; once peer selection is completed, we consider that any node in the peer set is equivalently suitable to store data.

a) Random Scheduling: The simplest scheduling choice, which is most commonly used in existing systems, amounts to just choosing a node at random within the peer set:

$$v_i(t) = 0.$$

Since all nodes will be tied in term of priority, scheduling will be chosen randomly. Random scheduling is extremely cheap and easy to implement because it is *stateless*: no information has to be kept about past node behavior.

b) Least Available First: A data transfer can stall if nodes that should receive the next pieces of data are not available. This strategy is based on assuming that nodes that have been online often in the past will continue to do so in the future; it thus makes sense to prioritize uploads towards nodes that have been less available in the past: when only high-availability nodes are online, data stored on them will be less likely to have already reached the maximum value m_i.

This scheduling policy observes past availability within a "window of past behavior" lasting for w timeslots:

$$v_i(t) = -\sum_{x=t-w}^{t} a_{i,x}.$$

c) Slowest First: This is a variant of the least-available-first policy, also taking into account the download speed of nodes, based on the idea that a node with slower download speed will complete receiving its maximal amount of data m_i in longer time:

$$v_i(t) = -\sum_{x=t-w}^{t} d_{i,x}.$$

d) Last Connected First: If the amount of time that nodes spend online is exponentially distributed, each node has the same probability of going offline independently of the amount of time spent online until the present. On the contrary, different distributions are observed in practice. In particular, if nodes that have been online for longer are more likely than others to remain online, it makes sense to prioritize uploads towards nodes that connected most recently, in order to capitalize on the capability of uploading to them before they disconnect:

$$v_i(t) = \max \{ x \in [1,t] : a_{i,t} = 0 \}.$$

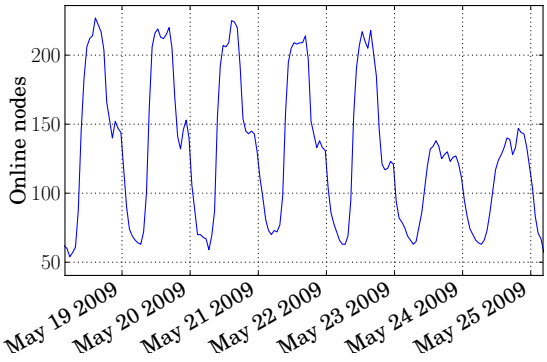

Figure 3: Online peers during a week in the trace.

e) Longest Connected First: If, as opposed to what has been discussed before, the amount of time a node spends online tends to be more concentrated towards the mean than in an exponential distribution, it makes sense to prioritize uploads towards node that got connected least recently:

$$v_i(t) = -\max\{x \in [1,t] : a_{i,t} = 0\}.$$

VI. Experimental Results

After showing how to compute optimal time to transfer and introducing scheduling strategies, we now focus on evaluating them, using simulations based on real network traces.

A. Simulation Settings

We perform our evaluation on real application traces, obtaining availability traces (*i.e.*, logon/logoff events) from an instant messaging (IM) server for a duration of 3 months, and dividing it in four 3-week periods. We argue that the behavior of regular IM users constitutes a representative case study for Peer-to-Peer storage applications. Indeed, in both IM and online storage, users are generally signed in for as long as their machine is connected to the Internet.

Adopting the same criteria used by the Wuala[3] online storage application, we only consider users that are online on average for at least four hours per day; this results in the trace of 376 users. Since we are especially focusing on situations where the peer set size is small, we consider the size of our traces to be sufficient for our goals. Availabilities are strongly correlated, in the sense that many users connect and disconnect around the same time. As shown in Fig. 3, there are strong differences between the number of users connected during day and night and between workdays and weekends.[4]

Uplink capacities of peers are obtained by sampling a real bandwidth distribution measured at more than 300,000 unique Internet hosts for a 48 hour period from roughly 3,500 distinct ASes across 160 countries [14]. To avoid cases where extreme bandwidth heterogeneity has a strong impact on data transfer completion time, we filtered out nodes with less than 1Mbps upload speed. To represent typical asymmetric residential Internet lines, we assign to each peer a downlink speed equal to four times its uplink,[5] capping it at 100Mbps.

Traces for a round of experiments with a peer set of n nodes is performed in this way:

1) A sample of n nodes is chosen from the period; availabilities are discretized in timeslots of 1 hour.
2) For each node i, upload and download speed is chosen from the bandwidth distribution as described above. In the simulation input, $u_{i,t}$ and $d_{i,t}$ are set to this value if the node is online, and to 0 otherwise.

In each experiment, we fix the values of o_i and m_i to be the same for each node i; we will therefore denote them simply as o and m. The object size o is set to 10GB. We do not consider the case where remote storage is a limitation, so we set $\overline{m}$ high enough to satisfy the constraint in any case.

Given that o and m are constant in each experiment run, there is a minimum number of peers $p = o/m$ needed to upload all the requested data per node. We will refer to this value in the following, in particular to evaluate the flexibility in data allocation which is expressed by the ratio n/p.

These simulation settings imply that optimal TTT will have the same distribution over all the simulation. A histogram of this distribution is plotted in Figure 4: our inputs determine long transfers often needing days to complete, with a skewed distribution due to differences in bandwidth and availability between nodes.

All our results are obtained by repeating the simulation over each of the 4 3-week periods for 10000 times in the scenario with no congestion and 100 times in the congested scenario. To account for a pessimistic case with maximal congestion, in the latter case all nodes start their transfer as soon as they connect for the first time.

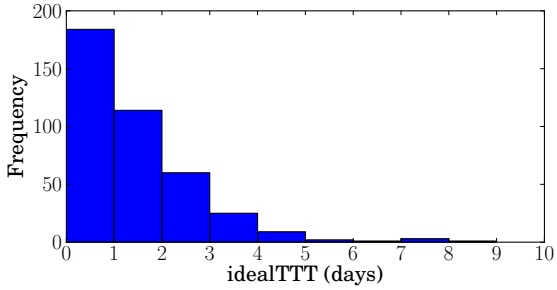

Figure 4: Histogram of idealTTT in our simulation settings.

B. Scheduling Strategies

We now take into account the scheduling overhead from Definition 6; Table III on the next page shows the overhead caused by the scheduling policies defined in Section V.

[3]http://www.wuala.com

[4]Our trace exhibits strong correlation in uptime because most users live in the same time zone; this impacts negatively both idealTTT and real TTT values in our experiments. Daily availability patterns are however present even in globally distributed applications, since neither human population nor the popularity of Internet applications are uniformly distributed across all timezones.

[5]At the time of this writing, Ookla's Net Index (http://www.netindex.com) reports an average download/upload bandwidth ratio of around 4.

Scheduling	w	Overhead
Random	–	12.66%
Least Available First	1 week	7.81%
Least Available First	1 month	7.76%
Slowest First	1 week	9.64%
Slowest First	1 month	9.69%
Last Connected First	–	9.53%
Longest Connected First	–	13.21%

Table III: Average scheduling overhead. No congestion, $m = 50MB$, $p = 200$, $n = 220$.

Random scheduling has a quite marked 12.66% overhead in TTT compared to optimal scheduling. We see the gap between optimal scheduling, least available first and random as a function of the information that can be inferred about future node uptime. Optimal scheduling can only be used in practice if future uptime can be forecast perfectly, while random uses no information at all. Other scheduling policies implement solutions justified by heuristic estimations about future node connectivity (*e.g.*, "nodes that remained online often in the past will continue to do so in the future" for Least Available First), and their performance reflects the accuracy and usefulness of these estimations. We consider estimating the degree of predictability of future node uptime in real traces, and the impact of this predictability, as an interesting open problem which is in our agenda for future work.

Between our simple scheduling strategies, Least Available First ranks best, reducing the overhead of random scheduling by an amount of around 5% of the total TTT, validating the fact that availability in the past is correlated to availability in the future, and that uploading to least-available nodes is a sensible strategy. Slowest First performs worse – even if better than random – reflecting the fact that bandwidth is less significant than availability, since during an online session it is generally possible to upload m bytes even to nodes with low bandwidth. Last Connected First also has better performance than random scheduling, reflecting the fact that nodes connected for longer will probably remain connected in the future; this is also validated by the fact that Longest Connected First performs worse than random scheduling.

We also observe that the length of the w window only has a very small impact on the quality of scheduling, as long as that window at least covers one week. In fact, shorter periods would suffer from weekly patterns such as users disconnecting for the week-end.

In Figure 5, we show that the scheduling overhead is distributed very unevenly across nodes: between 60% and 70% experience no overhead in TTT, while a minority of nodes experience the biggest difference. This is mainly due to the fact that nodes in our traces sometimes remain disconnected for whole days, strongly impacting their TTT.

C. Impact of Peer Set Size and Congestion

In Figure 6, we execute several instances of our experiments varying n and congestion settings, using a Least Available First scheduling strategy.

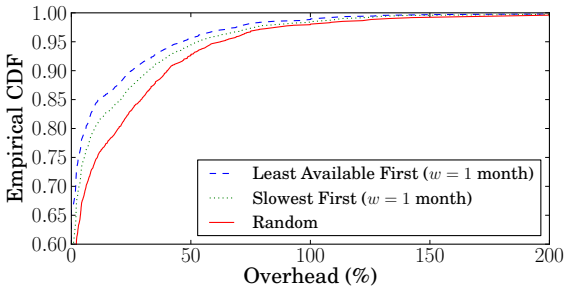

Figure 5: Cumulative distribution function of scheduling overhead. No congestion, $m = 50MB$, $p = 200$, $n = 220$.

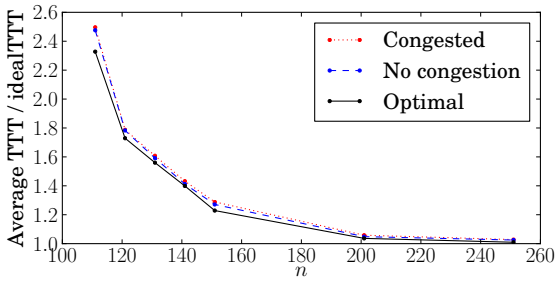

Figure 6: Evolution of TTT as peer set size n grows. $m = 100MB$, $p = 100$. Scheduling: Least Available First, $w = 1$ month.

A first key conclusion we draw is that the number n of nodes in the peer set is crucial to the performance: as n grows, TTT approaches the ideal TTT rather quickly. This raises an important message for P2P application designers: allowing flexibility in choosing where to upload data when scheduling uploads pays off handsomely in terms of performance.

A second conclusion is that congestion has an impact on time to transfer which is almost negligible, and definitely smaller than the scheduling overhead. This is due to the fact that in our simulation the upload bandwidth of data owners is almost always the bottleneck; as such, in the case of several simultaneous uploads, we think it is very unlikely that they would slow down noticeably the transfers.

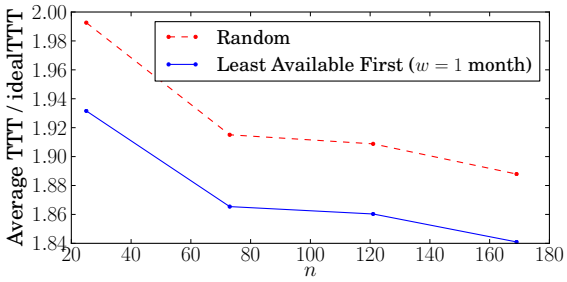

Figure 7: Evolution of TTT with constant $n/p = 1.2$. Congested case.

Scheduling	Nodes completed within optGTTT
Random	71.33%
Least Available First (w =1 week)	73.11%
Slowest First(w =1 week)	72.34%
Last Connected First	72.26%

Table IV: Congested case. $n = 24$, $p = 20$, $m = 500MB$.

Scheduling	w	Overhead
Random	–	1.43%
Least Available First	1 week	1.13%
Least Available First	1 month	1.06%
Slowest First	1 week	1.38%
Slowest First	1 month	1.37%
Last Connected First	–	1.20%
Longest Connected First	–	1.70%

Table V: Scheduling overhead for downloads. No congestion, $m = 50MB$, $p = 200$, $n = 220$.

What we pointed out raises a further question: if the number of nodes in a peer set cannot be increased above a certain threshold, would it be possible to still obtain good performance by lowering the number of required peers p? In Figure 7 on the preceding page, we show the ratio between TTT and idealTTT keeping the n/p ratio constant. In this case, the performance is only slightly sensitive to the number n of nodes: as n decreases, TTT grows moderately. In fact, this reiterates our point above: the flexibility in choosing the nodes to upload data to is key in our case. On the other hand, if the network size grows while keeping the "flexibility factor" n/p constant, there is only a small impact on the overhead due to scheduling.

D. Optimal Congested Case

We now consider how our experimental results compare to the optimal time to complete all transfers in the case of congestion, optGTTT. We recall that we considered an extreme situation where each node uploads data at the beginning of the simulation. Unfortunately, due to the sheer size of the linear programming problem we defined,[6] we could only manage to solve a system with n up to 24.

With optimal scheduling in the congested case, all nodes would be able to complete their transfers within optGTTT. In Table IV, we report on how many actually manage to complete their data transfer in this time. A relevant percentage of nodes (around 27-29%) complete their transfer after optGTTT. We stress that this loss is only due to scheduling inefficiency, since an optimal schedule would complete all the data transfers within optGTTT. We attribute the relatively high number of nodes completing their transfer later to the skewness in the distribution of scheduling overhead already observed in Figure 5: a minority of nodes are heavily affected by the scheduling overhead, and this makes them "miss" the optGTTT deadline.

E. Downloads

As a last experiment, we repeated our experiments for the case of downloads as defined in Section II-B. We maintained all the inputs that we had for other experiments, switching the download and upload bandwidth of nodes, since now data is transferred in the opposite direction. We do not take in consideration congestion in this case: the case where multiple nodes require downloading the same piece of content from the same node is well known and generally solved with content delivery solutions such as BitTorrent [15]. The case of downloads is interesting also in the case of application use cases where the peer set size is very large, making the impact of scheduling in uploads basically irrelevant: in this case, the n/p ratio refers to the redundancy rate applied to uploaded data; a redundancy value high enough to make scheduling irrelevant could be unpractical to obtain in many situations.

Due to asymmetric bandwidth, scheduling choices are much less relevant in the case of downloads: in fact, a node will typically be able to saturate the upload bandwidth of various peers at once: a real scheduling decision will only happen when many uploaders are available to send data simultaneously to the recipient; in this case, the transfer will likely be completed soon anyway. In Table V, we repeat the measurement of Table III in this new setting; we obtain scheduling overheads of around one order of magnitude less than in the upload case. It is interesting to note that the relative performance of scheduling policies remains unchanged: Least Available First stills fares better than all other alternatives.

VII. Related Work

Data transfer scheduling is a topic that has not been explored much in the literature on P2P storage. In the only other piece of research on the topic we are aware of, Birk and Kol [16] analyzed random scheduling by analytically modeling peer uptime as a Markovian process. In that respect, we confirm their finding: the completion time of random scheduling converges to to the optimal value as the system size grows.

Simple mathematical models such as Markov chains can be treated analytically, but they cannot capture the inherent complexity of connectivity patterns that are ultimately due to the behavior of the human users that run the P2P applications. As such, when churn is abstracted through a model, any analysis of the scheduling policy risks dealing with the artifacts of models rather than with real-world properties of traces. In contrast to them, our approach allows estimating the performance of scheduling policies in realistic cases, and the comparison with optimal scheduling computed *a posteriori* allows us to estimate the further efficiency gain that could be achieved with more sophisticated scheduling policies.

Several regularities in uptime behavior are known to exist [9], [17], including uptime correlation and diurnal and weekly patterns. In this work, we acknowledge these regularities, using real-world traces as inputs, and we have shown how simple scheduling techniques can exploit them in order to obtain better performance.

As we have shown, scheduling matters especially when the peer sets are rather small. In literature, several reasons exist to restrict the size of peer sets. A few examples are:

[6]Recall from Definition 11 that we obtain a constraint matrix of size $n\,(2T + n + 2) \times n^2T$.

- When distributed hash tables (DHTs) are used, data should be stored on nodes which are easily retrievable using the DHT lookup functionality. The smaller the peer set is, the easier the data is to find. Examples of such systems are Pastry [2] and Pastiche [3].
- To avoid storing data on untrusted nodes, data gets only sent to machines owned by friends of the data owner, in order to leverage on trust relationships between users. FriendStore [6] and Safebook [7] are examples of systems adopting this solution. The bigger the peer set is, the weaker trust links need be.
- Nodes in a peer set can be chosen in order to guarantee different system properties, for example to ensure high availability with lower redundancy [4], or to create incentives for cooperation by grouping nodes with similar characteristics [5], [18]. In these cases, larger peer sets can undermine the benefits of these peer selection rules.

A further reason why peer sets are restricted stems from the fact that nodes have to manage a large number of data blocks. If each of them is stored on different neighbors, "bookkeeping" (*i.e.*, keeping track of connectivity patterns and data loss episodes) becomes very expensive; this motivates putting several data blocks on the same peers. The most widely deployed online storage application adopting a peer-to-peer solution, Wua.la, stores data on a fixed set of 130 peers per node [19].

We have shown that even a moderate increase of peer set size can result in definitely better data transfer performances. Investigating the trade-off with the fact that systems can behave better when smaller peer sets are chosen is in our plans for future research.

VIII. Conclusions

In this work, we introduced the problem of scheduling large data transfers for Peer-to-Peer storage applications. We described the tight coupling between scheduling strategies and characteristics of traces, and we underlined the importance of using real availability traces rather than relying on a mathematical abstraction of node churn. We formalized the notions of optimal scheduling strategy and of scheduling overhead, *i.e.* the gap between the adopted scheduling strategy and the optimal scheduling evaluated *a posteriori*. We showed how to compute this optimal scheduling value using polynomial algorithms, using a max-flow formulation for the case with no congestion and linear programming in the congested case.

Via simulation, we obtained various insights. First, as the number of nodes in peer sets grows, the time to complete transfers decreases rapidly. This is an important message to application designers: allowing a small degree of flexibility with respect to the choice of nodes to adopt in the overlay for storing data pays off significantly. Second, a simple scheduling policy such as Least Available First manages to cut around 40% of the scheduling overhead, based on the assumption that nodes that have been online often in the past will continue to do so in the future. Third, the overhead with respect to optimal scheduling is very unevenly distributed: many nodes will barely experience a difference between the optimal schedule and the one they took in practice, but for a relevant percentage of them there is the possibility that bad schedule choices will result in much longer data transfer times. Fourth, when many nodes are uploading data at the same moment, congestion only has a small impact on transfer completion times. Fifth, the impact of scheduling in the case of downloads is much less significant, due to the asymmetry of bandwidth on nodes.

The performance of scheduling algorithms is a consequence of the predictability of the connectivity patterns of users. If their connections and disconnections could be forecast with certainty, there would be a way to devise optimal scheduling; our simple scheduling policies exemplify heuristic "guesses" on future node connectivity, and they manage to reduce the time needed to complete transfers. We are currently investigating more sophisticated techniques to predict user availability, and we plan to apply them also to the scheduling problem.

References

[1] L. Toka, M. Dell'Amico, and P. Michiardi, "On scheduling and redundancy for p2p backup," *Arxiv preprint arXiv:1009.1344*, 2010.

[2] A. Rowstron and P. Druschel, "Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility," *SIGOPS Oper. Syst. Rev.*, vol. 35, no. 5, pp. 188–201, 2001.

[3] L. Cox and B. Noble, "Pastiche: Making backup cheap and easy," in *USENIX OSDI*, 2002.

[4] L. Pamies-Juarez, P. García-López, and M. Sánchez-Artigas, "Heterogeneity-aware erasure codes for peer-to-peer storage systems," in *IEEE ICPP*, 2009.

[5] L. Pamies-Juarez, P. García-López, and M. Sánchez-Artigas, "Rewarding stability in peer-to-peer backup systems," in *IEEE ICON*, 2008.

[6] D. Tran, F. Chiang, and J. Li, "Friendstore: cooperative online backup using trusted nodes," in *ACM SocialNets*, 2008.

[7] L. Cutillo, R. Molva, and T. Strufe, "Safebook: A privacy-preserving online social network leveraging on real-life trust," *Communications Magazine, IEEE*, vol. 47, no. 12, pp. 94–101, 2009.

[8] D. Stutzbach and R. Rejaie, "Understanding churn in peer-to-peer networks," in *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pp. 189–202, ACM, 2006.

[9] S. Le Blond, F. Le Fessant, and E. Le Merrer, "Finding good partners in availability-aware p2p networks," *Stabilization, Safety, and Security of Distributed Systems*, pp. 472–484, 2009.

[10] M. Luby, "LT codes," in *Foundations of Computer Science, 2002. Proceedings. The 43rd Annual IEEE Symposium on*, pp. 271–280, IEEE, 2002.

[11] A. V. Goldberg and R. E. Tarjan, "A new approach to the maximum flow problem," in *ACM STOC*, 1986.

[12] L. Khachiian, "Polynomial algorithms in linear programming," *Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki*, vol. 20, pp. 51–68, 1980.

[13] N. Karmarkar, "A new polynomial-time algorithm for linear programming," in *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pp. 302–311, ACM, 1984.

[14] M. Piatek, T. Isdal, T. Anderson, A. Krishnamurthy, and A. Venkataramani, "Do incentives build robustness in bittorrent," in *USENIX NSDI*, 2007.

[15] B. Cohen, "Incentives build robustness in BitTorrent," in *Workshop on Economics of Peer-to-Peer systems*, vol. 6, pp. 68–72, Citeseer, 2003.

[16] Y. Birk and T. Kol, "Coding and scheduling considerations for peer-to-peer storage backup systems," in *SNAPI*, IEEE, 2007.

[17] M. Steiner, T. En-Najjary, and E. Biersack, "Long term study of peer behavior in the KAD DHT," *Networking, IEEE/ACM Transactions on*, vol. 17, no. 5, pp. 1371–1384, 2009.

[18] P. Michiardi and L. Toka, "Selfish neighbor selection in peer-to-peer backup and storage applications," in *Euro-Par*, 2009.

[19] T. Mager, "Measurement study of wuala, a distributed social storage service," Master's thesis, Eurecom and TU Darmstadt, 2009.