Transaction rate limiters for peer-to-peer systems*

Marcos K. Aguilera† Mark Lillibridge Xiaozhou Li

Hewlett-Packard Laboratories
1501 Page Mill Rd
Palo Alto, CA 94304, USA
firstname.lastname@hp.com

Abstract

We introduce transaction rate limiters*, new mechanisms that limit (probabilistically) the maximum number of transactions a user of a peer-to-peer system can do in any given period. They can be used to limit the consumption of selfish users and the damage done by malicious users. They complement reputation systems, solving the* traitor problem. *We give simple distributed algorithms that work over time frames as short as seconds and are very robust: they use no trusted servers and continue to work even when attacked by a large fraction of users colluding. Our algorithms are based on a new primitive we have devised,* probably-anonymous queries*, which guarantees anonymity with a specified probability.*

1 Introduction

Peer-to-peer systems can be a convenient venue for parties to exchange goods and services. In such systems pairs of users transact with each other; for example, one user may download a song from or agree to store a document for another user. Each user has a *natural transaction-rate limit*, the maximum rate at which she can do transactions. This rate is usually determined by physical limitations like her network bandwidth or processing power, but sometimes is determined by partner availability. Unfortunately, the natural rate is often undesirably high, especially for malicious users.

Accordingly, we propose *transaction rate limiters*, or simply rate limiters, which are mechanisms to (probabilistically) limit the transaction rates of users. Each user can be forced to transact as slowly as desired, with different users having different limits that can vary over time. Our transaction-rate limiter implementations can work over time frames as short as seconds and are extremely robust: they do not rely on trusted servers, which may be easy to take out or subvert, and continue to work even when attacked by a large fraction of the system's users colluding.

Transaction rate limitation is important both to limit selfish users from consuming far more than their fair share of resources (e.g., limit songs downloaded/hour) and to limit how fast malicious users can do damage (e.g., limit messages sent/minute to limit spam). The need for rate limitation is not removed by using reputation systems because they take so long to act: we expect any decentralized robust reputation system to act on a substantially slower timescale than our limiters because it must gather information from a substantial fraction of nodes while our limiter need talk to only a few nodes before each transaction. The slowness of reputation systems makes them vulnerable to attacks by *traitors*, dishonest users that act honestly for some period to build up their reputations and later quickly engage in many, many dishonest transactions [14].

Adding a rate limiter to a reputation system prevents traitors from doing significant damage before they are expelled. In addition, it provides defense in depth: even if an attacker figures out how to fool the reputation system, the much simpler (and hence more likely to be correct) limiter will limit the damage. Rate limiters are useful even without reputation systems, although more damage can occur in that case because malicious users that constantly do a small amount of damage cannot be expelled. We are unaware of any reputation systems that work under the severe conditions (e.g., large coalitions of dishonest users) that our limiters can tolerate so using transaction rate limiters may be the only available option for systems that must operate under such conditions.

Briefly, our rate limiters work roughly as follows. To limit users' transaction rate to k/T, we split time into periods of duration T, limit each user to transacting with at most

*Reproduced from the *Proceedings of the International Conference on Peer-to-Peer Computing 2008* (P2P'08);

†Author's current affiliation: Microsoft Research Silicon Valley

k honest users per period, and have each honest user transact with any given user at most once per period. The challenge here is finding a way for an honest user Alice to avoid becoming the $k+1$th user a dishonest user Bob is transacting with given that (1) she cannot see who he is talking to, (2) there may not be enough time for Alice to contact all the other users, and (3) there are no users she can be sure she can trust.

The key idea in our scheme is to force Bob to tell her who he is transacting with by having all the honest users that wish to transact with him (including her) anonymously and simultaneously ask Bob who he is transacting with. If Bob wants anyone to do business with him, he must give out their name: any user who receives back a list without their name on it or containing more than k users knows Bob is trying to cheat and refuses to do business with him. Unfortunately for Bob, because he has no control over who gets each name, the more people he tries to do business with, the more likely each of them catches him cheating. We show in Section 5.2 that the expected number of honest users that deal with him under these circumstances is bounded above by k. Our limit is thus probabilistic.

Unfortunately, in a pure peer-to-peer system it is difficult to implement always-anonymous communication, especially if there is a large number of dishonest parties colluding with Bob. We therefore instead use *probably-anonymous queries*, which are guaranteed to be anonymous only with some probability p. We implement probably-anonymous queries by using random relays and assuming a limit on the fraction of dishonest users. The probability that a sender's identity is exposed (at most $1-p$) need not be negligible, so in our transaction limiters users compensate by making several queries to Bob so that the probability that all such queries are not anonymous is small. We can precisely quantify how p affects the ability of Bob to cheat, so that we can choose an appropriate number of queries to bound the amount of cheating.

This paper is organized as follows: in the next section, we introduce an example of how our limiters might be used. In Section 3, we explain our system model. In Section 4, we describe probably-anonymous queries and how to implement them. In Section 5, we describe and analyze our basic transaction rate limiter implementation. In Section 6, we estimate key performance parameters for our example. In Section 7, we describe how to handle churn. In Section 8, we briefly discuss other extensions of our work. Finally, in Section 9 we discuss related work and in Section 10 our conclusions.

2 Example: Limiting audio spam

Consider an imaginary decentralized peer-to-peer system, which we shall call Musictella, that is loosely based on its namesake, Gnutella. Like Gnutella, Musictella's goal is to allow its users to share songs. Each user makes available a set of songs that they are willing to share. Other users can then request and receive those songs. We will not be concerned in this paper with how users find song providers, assuming only that there is some mechanism for interested users to locate other users that have songs they want.

Without defense mechanisms, Musictella is likely to suffer from spam: spammers may attempt to share audio ads with as many people as possible by mislabeling their ads as popular songs. More problematically, Musictella may be attacked by large organizations that want to make Musictella (nearly) unusable by spamming the system with fake or bad music. While it may be possible to programmatically distinguish audio ads from songs, it is unlikely that bad music can be reliably distinguished (e.g., an attacker might simply send the wrong song).

Accordingly, any effective defense against spamming must limit spammers' ability to send spam. Ways to do this include limiting the number of users a node can distribute to at once (e.g., transaction rate limiters) and banning spammers based on users' complaints (e.g., reputation systems).

Either method requires holding down the number of system nodes controlled by spammers at any given time. How best to do this is beyond the scope of this paper. For concreteness, we shall assume that Musictella limits malicious nodes by requiring that each node that wishes to join or renew its membership solve a new hard computational puzzle and by assuming that Musictella has a large enough user base so that spammers do not have enough computational power to control more than a small fraction of Musictella's nodes at any time [15]. In our examples, we will assume that Musictella has one million nodes, of which at most 1% are controlled by spammers at any one time. See the literature on Sybil attacks [2, 5, 20, 22] for more discussion of this problem and methods for solving it.

We shall now consider and contrast three approaches for controlling spam: (1) using a transaction rate limiter, (2) using a reputation system, and (3) using both.

Using a limiter: Adding a transaction rate limiter to Musictella is straightforward: treat each song transfer as a transaction between a provider and a receiver and limit the rate of transactions each provider can do. Because the same node can be both a provider and a receiver in this example, we will need to limit user-role pairs rather than users in order to avoid limiting how fast a node can receive songs as well. One simple way to do this is to split each node into two virtual nodes, one of which is allowed only to send and one of which is allowed only to receive; we then limit only the provider virtual nodes.

Suppose the provider limit is say 10 songs per hour and each non-spammer node provides one song per hour.[1] Then

[1] The assumed sharing rate may be unrealistically high: most Gnutella

in one hour the spammers will send $10 \cdot 1\% \cdot 1$ million = 100,000 pieces of spam and the honest users will send $1 \cdot 99\% \cdot 1$ million = 990,000 good songs so 90.8% of received songs will be spam free. In the absence of any limiters or banning, the percentage of received spam will depend on the natural-transaction rate limit of the attacker, which could easily be as high as 360 songs/hour per node (10 Mb/s bandwidth, 10 MB song size) yielding only 21.6% spam-free songs.

Using reputations: A simple reputation system for Musictella might work as follows: when users discover by listening to a "song" that they have been given spam, they broadcast a complaint against the node that provided the spam; once a user hears enough complaints about a given node, they stop transacting with or playing songs from that node. How many complaints are enough? Because the attacker, which could control 10,000 nodes, may try to falsely accuse non-spammers in order to stop the flow of illegal music, 10,001 complaints are needed to be sure a node is bad.

The Achilles' heel of this reputation system is the complaint broadcast system, which must be both very fast and acceptably efficient. It needs to be fast because if the attacker has each of his nodes take turns using all his bandwidth (i.e., all of the attacker's machines are pretending to be a single node at a time), then the current spamming node can send $360 \cdot 1\% \cdot 1$ million = 3.6 million songs per hour $= 1,000$ songs per second, which in 10 seconds blows past the maximum 10,001 spams that an ideal reputation should allow any node to send. Every second beyond that that banning takes allows each spammer node to spam an additional 1,000 times.

It is hard for the broadcast system to be efficient because banning one node requires at least $O((1-\theta)n^2)$ communication where n is the number of system nodes, θ is the constant fraction of nodes that are honest, and $(1-\theta)n$ is the number of complaints required for safe banning. (Our limiters, by contrast, we will show in Section 6 add only $O(\log n)$ communication per transaction.)

Using both: Transaction rate limiters can complement reputation systems. Here adding a transaction rate limiter to the reputation system described above mitigates the broadcast problem because broadcasts can then be much slower while allowing the same amount of spam through. Communication efficiency can be better in practice as well because messages can be batched and combined. Without limiters, such a reputation system might never work in practice.

In addition to the spamming problem, the designers of Musictella may also wish to increase the fairness of their system by limiting how much of the system's resources any one user can consume at a time. They could do this by using a second transaction rate limiter to limit how many songs any given user can receive per hour. The second limiter's setup is similar to the anti-spamming one but we instead limit the rate of transactions each receiver node can perform to say 5 songs per hour. In general when multiple limiters are used, each limiter can limit a different class of users or user-role pairs to a possibly different rate limit. Thus, for example, when using limiters in a system for purchasing electronic goods, we could limit buyers, sellers, or both.

nodes share no songs. If so, the limit should be proportionately reduced.

3 System assumptions

In order to simplify the presentation of our algorithms, we are going to assume (unrealistically) for the next several sections that there is no churn, nodes are always connected, and each node knows the identity of every other node in the system. Later, in Section 7 we describe how to handle churn, temporarily unavailable nodes, and strangers.

For now, we consider a distributed peer-to-peer system with a fixed large set Π of users (equivalently, nodes). Each user is either permanently *honest*, meaning that she always follows our protocols, or *dishonest*, meaning that she may deviate from our protocols. A fraction $\theta > 0$ of users are honest. Dishonest users may collude towards a common goal.

Nodes can communicate by sending messages over reliable links, which do not lose, duplicate, or create messages (e.g., a TCP connection). Eavesdropping is not possible—messages are visible only to their senders and receivers—and communication is probabilistically synchronous: there are constants $t_d>0$ and $p_d>0$ such that, with probability at least p_d, a message is received within time t_d of being sent. In practice, t_d is of the order of hundreds of milliseconds and p_d is close to 1. For example, Hu et al. [7] measured that 90% of the round trip delay time (RTT) from any of three server replicas (located in the US, Europe, and Asia) to a large number of destinations (about 140,000) fall below 400 milliseconds. In fact, p_d can be made as close to 1 as desired (at the cost of increasing t_d) by retransmitting messages.

Users can only send messages to users whose names they know (including users they have received messages from). We assume that each user $u \in \Pi$ knows a (possibly different) subset K_u of users. Which users in K_u are honest is not known to (honest) u. For now, we will assume the system's membership is public (i.e., $K_u = \Pi$ for every u).

Nodes have nearly-synchronized clocks, such as those obtained by using Network Time Protocol (NTP) servers or (less accurately) manual synchronization. Nodes' clocks increase monotonically and differ by at most $\pm\epsilon$ from real time at any given time. In Wide Area Networks (WANs), NTP usually can keep ϵ down to tens of milliseconds [16]. For simplicity, we assume computation takes negligible time.

For the reader's convenience, Figure 1 lists the symbols

symbol	description
Π	set of all system users
K_u	users known by user u
θ	fraction of honest users
t_d	message transmission time
p_d	probability a message is delivered within t_d
ϵ	max. time any local clock differs from real-time
p	probability a query succeeds
q	probability a query fails
t_r	thinking time allowed before must respond
t_Δ	time a query requires independent of t_r
r	probes per query
k	number of transactions allowed per period
T	size of a period
δ	upper bound on extra transactions a dishonest user can do per period

Figure 1. Symbols defined in Sections 3, 4, and 5 respectively.

used throughout this paper along with their concise definitions.

4 Probably-anonymous queries

To implement transaction rate limiters, we use probably-anonymous queries. A *(p, t_Δ)-probably-anonymous query* is a communication primitive that takes arguments m, v, and t_r. This primitive allows an honest user u to send query m to (honest or dishonest) user v and v to optionally send one answer to u after at most t_r of thinking time. It ensures, with probability p, that (a) v receives the query m; (b) v learns nothing about who sent m other than what m may convey and possibly u's local time when u made the query; (c) if v replies with exactly 1 answer within real-time t_r of receiving m, then u receives that answer; (d) u receives exactly 1 answer, either one from v or "timeout"; and (e) u receives an answer within $t_\Delta + t_r$ of the query's start by his clock.

Probably-anonymous queries can be implemented in a highly robust peer-to-peer fashion by using a random node as a relay to anonymize communication as shown in Figure 2. We use nonces (n_u and n_w) to distinguish multiple instances of the protocol running simultaneously and to ignore messages with spoofed sender information.

Consider the case where no protocol messages are late (i.e., take longer than t_d) and w is an honest user. Because u and w's clocks can differ by at most 2ϵ, w receives m by $t_1{+}t_d{+}2\epsilon$ his time. Note that the time he forwards m, $t_1{+}t_d{+}2\epsilon$ *his time*, depends only on t_1 (the local time u made the query) and w. Thus v on receiving (m, n_w) learns only m, n_w, w, and (roughly) t_1. Knowledge of t_1 and m are expressly permitted by (b); n_w is a fresh random number and thus conveys no information about u's identity. Because by the public membership assumption every user chooses its relays from the same set, Π, the choice of w reveals nothing about the identity of u either, so (a) and (b) hold.

u queries v with m allowing think time t_r and gets an answer:

u: $w \leftarrow$ random user from set K_u;
  $t_1 \leftarrow$ current time, $n_u \leftarrow$ random number;
  send (m, v, t_r, t_1, n_u) to w

w: receive (m, v, t_r, t_1, n_u) then wait until $t_1 + t_d + 2\epsilon$;
  $n_w \leftarrow$ random number;
  send (m, n_w) to v

v: receive (m, n_w) then choose an answer a;
  send (a, n_w) to w

w: wait until $t_1 + 3t_d + t_r + 4\epsilon$;
  if have received (r, n_w):
    then $a' \leftarrow r$
    else $a' \leftarrow$ "timeout"
  send (a', n_u) to u

u: receive (a', n_u);
  return a'

Figure 2. Implementing probably-anonymous queries with $p = \theta p_d^4$ and $t_\Delta = 4t_d + 6\epsilon$.

If v replies once with (a, n_w) after a real-time delay of at most t_r, then w will receive the reply at most real-time $2t_d + t_r$ after forwarding m to v. The local time w receives the reply at is thus at most $t_1{+}3t_d{+}t_r{+}4\epsilon$, with the extra 2ϵ due to the possibility that w's clock speeded up relative to real-time during the interval. Accordingly, a' will be set to a and u will receive a as the query answer so (c) holds. Regardless of whether v replies, w will send (a', n_u) to u at $t_1{+}3t_d{+}t_r{+}4\epsilon$ w's time where a' is either "timeout" or an answer from v. This message can arrive no later than $t_1{+}4t_d{+}t_r{+}6\epsilon$ u's time. Thus, (d) and (e) hold for $t_\Delta = 4t_d{+}6\epsilon$.

Thus for the case we have considered, which has probability $p = \theta \cdot p_d^4$, (a) through (e) hold for $t_\Delta = 4t_d{+}6\epsilon$. Figure 2 thus implements a (θp_d^4, $4t_d{+}6\epsilon$)-probably-anonymous query.

5 Transaction rate limiters

We assume time has been divided into consecutive periods of duration T and that a distinguished subset of users, $\mathcal{L}$, has been designated as users that need their transaction rate limited. A k-rate limiter acts to limit the expected number of transactions each limited user $\ell \in \mathcal{L}$ can do with honest users in any given period to k. Each limited user can thus perform transactions at a rate of at most k/T.

Our rate limiters work by having each user u wishing to transact with a limited user ℓ in a given period run an ap-

approve u transacting with limited user ℓ?:

u: wait until start t of given period;
probably-anonymous query ℓ with t allowing think time 0
in parallel r times

ℓ: answer all queries t with the name of the user with whom ℓ plans to transact in period $[t, t+T]$

u: wait until $t + t_\Delta + 2\epsilon$;
if all answers returned before $t+t_\Delta$ are u
then return "approve"
else return "reject"

Figure 3. Basic approval protocol for a $1+\delta$-rate limiter where δ depends on parameter r.

proval protocol with ℓ first; u proceeds with at most one transaction with ℓ if the approval protocol gives permission. If u is also limited, then both u and ℓ run the approval protocol with each other in parallel and the transaction proceeds only if both protocols approve.

We assume each limited user has already decided by the start of a period who he wishes to transact with in that period. How exactly this occurs does not matter for our algorithms; for example, ℓ may take reservations from users that come to him, or ℓ may solicit users, or both. If there is a concern that some users may be starved of service (e.g., never get to transact with a given node), then a method should be chosen that ensures that any user wanting service from a given node eventually gets it. Such methods include holding lotteries among current applicants and taking applicants first-come, first-served off a waiting list.

For simplicity we implement a 1-rate limiter (more precisely a $1+\delta$-rate limiter, $\delta \ll 1$); a k-rate limiter (really $k+k\delta$) can be built from this implementation by allowing each limited user to run k instances of the $1+\delta$-rate limiter in parallel: each user wishing to transact with ℓ is assigned one of these instances by ℓ, and each instance runs independently of each other.

Alternatively, our $1+\delta$-rate limiter can be generalized (not shown) to a $k+\delta$-rate limiter by having ℓ answer queries with a list of the names of the at most k users he intends to transact with in that period and by having approval occur only if all the answers u receives in time are lists of length at most k containing u. Note that because the approval protocols of k-rate limiters take k as a parameter, it is easy to build more complicated limiters where the limits vary from user to user and from period to period; for example, new nodes might have more stringent limits than older or more reputable nodes.

5.1. Basic approval protocol

Figure 3 shows the approval protocol for our basic $1+\delta$-rate limiter. Here u attempts to anonymously ask ℓ to return the single user with whom ℓ intends to transact; if an answer other than u is received, the protocol disapproves. Because we can ask only probably-anonymous queries, which may have a high failure rate, we have to resort to multiple queries to ensure a sufficiently high probability that at least one of them is truly anonymous. Even with r queries, there is still some chance of failure, potentially allowing ℓ an extra δ transactions; however, as we shall see, by increasing r we can make δ as small as desired.

To prevent ℓ from distinguishing among the anonymous queries of the users that wish to transact with him in a given period, all users do their queries at the same time t on their local clocks (recall that queries can leak this time). At the end of the protocol, u waits an extra 2ϵ before returning to ensure he does not subsequently take actions (outside our protocol) that cause ℓ to learn the result of u's query before another user's query has finished, which might allow ℓ to make his answer to the second query depend on the result of the first query.

5.2. Analysis

Suppose n honest users wish to transact with ℓ. Let H be the expected number of these that receive approval. Together, the honest users will make rn probably-anonymous queries, some of which may turn out anonymous and hard for a dishonest ℓ to answer. If we pretend for a moment that each honest user makes exactly 1 query and that it succeeds (i.e., (a)–(e) holds), then we claim that H is bounded above by 1: ℓ has n indistinguishable queries to answer, each of which he must answer with that inquirer's name in order to get that inquirer to transact. The expectation that an answer matches an inquirer is $\frac{1}{n}$. Thus, by the linearity of expectation, no matter how ℓ answers, the expected number of matches is at most $n \cdot \frac{1}{n} = 1$. Additional queries cannot raise H because they do not help ℓ answer the successful queries.

Unfortunately, there is a probability that all r queries of a user are non-anonymous. This probability is at most q^r where $q = 1-p$. If we conservatively assume that a dishonest ℓ can trick such users into transacting, then an upper bound on H is $1 + nq^r$. Here 1 bounds the number of approved users that made at least one successful query and nq^r bounds the number of approved users that did not.

In practice, a dishonest ℓ can nearly achieve this limit, for example, by adopting the following strategy: choose some user u_0 who wishes to transact then answer all anonymous queries with u_0's name, and all non-anonymous queries with their asker's name. With this strategy, ℓ can get u_0 to transact plus any other user whose r queries fail to be anonymous. If we assume all messages arrive in time (i.e., $p_d = 1$) and ℓ learns the asker of a query with probability q, then $H = 1 + (n-1)q^r$.

A dishonest ℓ's advantage over an honest ℓ is at most $\delta = nq^r$ transactions with honest users, where n is bounded by the number of users in the system, $|\Pi|$. δ can be made as small as desired by setting $r = \log_q \frac{\delta}{|\Pi|} = \log_{q^{-1}} (|\Pi|\delta^{-1})$.

6 Performance

To get a sense of how our transaction rate limiter might perform in practice, we now estimate for the Musictella example of Section 2 the key performance numbers, namely the number of extra messages that must be sent per transaction and the latency added by using the limiter.

Per Section 2, the system has $|\Pi| = 10^6$ real nodes, of which at least a fraction $\theta = 0.99$ are honest; moreover, virtual provider nodes are supposed to be limited to at most 10 transactions per hour (equivalently one transaction per six minutes). We shall assume that an honest node can always respond to a probe within $t_r = 1$ second. Per Section 3, we shall assume that the probability of a message being received within $t_d = 1$ second is at least $p_d = 0.95$. Moreover, we will assume that NTP holds the time difference between any two nodes to within $2\epsilon = 100$ milliseconds.

If we set $k = 1$ and $T = 6$ minutes, then honest users will be limited to sending 10 songs per hour as desired. Dishonest users, however, will be able to exceed this by an additional 10δ songs per hour, where $\delta = mq^r$, m is the maximum number of users that a dishonest user can interest in receiving its songs in a single period, $q = 1-p$ is the probability a given probably-anonymous query fails, and r is the number of probes each node does per transaction. In the worst case, $m = |\Pi|$, its maximum value (perhaps everyone wants a new, not yet released song). For our probably-anonymous query implementation with $p_d = 0.95$, we have $q = 1-p = 1 - \theta p_d^4 = 0.194$.

If we choose to limit 10δ to 0.1 (e.g., a dishonest user can send at most 10.1 songs per hour), then we can pick $r = 12$: per Subsection 5.2, we need $r \geq \log_{q^{-1}} (|\Pi|\delta^{-1})$ and $q^{-1} = 5.15$. Equivalently, we could have used the same δ but chosen $T = 6.06$ minutes instead so dishonest users could send at most 10 songs per hour while honest users could send only at most 9.9 songs per hour.

Each node that wishes to receive a song must then make 12 probes, which results in $4r = 48$ messages being sent. This number can be reduced somewhat by increasing t_d (and hence p_d); for example, $t_d = 10$ seconds might make $p_d = .99$ and hence $q = 0.49$ and $r = 7$. Alternatively, instead of changing t_d, we can make δ larger: $10\delta = 1$ yields $r = 10$, for example.

The added latency due to the probing process is $t = t_\Delta + t_r + 2\epsilon = 4t_d + t_r + 8\epsilon = 5.4$ seconds (remember that all probes for a given period are done in parallel). For comparison, downloading a typical MP3 song (4 MB) over a 1 Megabit per second link takes at least 32 seconds.

If we keep everything except the number of nodes constant, we find that we need $O(\log m) \leq O(\log |\Pi|)$ probes per attempted transaction and $O(1)$ extra latency. Increasing the fraction of dishonest nodes also increases the required value of r:

dishonest fraction $(1-\theta)$	required probes (r)
0.1%	11
1%	12
10%	14
25%	20

Even the largest r here (20) yields a number of messages (80) that is small compared to the number of packets required to download a typical MP3 song. If we use IP addresses as names and 8-byte values for the other fields, each message has at most 32 bytes, which means our limiter uses negligible bandwidth.

There is another source of latency in our system: waiting for the start of the next period. If Fred wants to download a song from Sally now and Sally has no other requests, he will have to wait three minutes on average for the start of the next period. The extra time Fred may have to wait for Sally to serve earlier requests due to the imposed rate limit is not overhead due to our transaction rate limiter implementation, but rather inherent in any approach that limits rates.

In many cases, it may be possible to hide these overhead latencies by doing the probing/waiting in parallel with the transaction. For example in Musictella, Fred could start downloading his song as soon as Sally has free bandwidth but only look at the song after the limiter has given approval (the download could be abandoned early on a rejection). We are also working on more complicated limiters (not otherwise described in this paper) that trade making more probes for shorter periods. Very roughly, they work by having Fred decide in period i and then *spoil* periods $i+1$, $i+2$, $\ldots i+l-1$ for Sally; Fred spoils a period by probing and then contacting and warning off anyone whose name he receives (other than his). This effectively allows Sally to only do $\frac{k}{lT}$ transactions per period so T can be made $1/l$ smaller.

7 Churn

In order to make our system work with a changing system membership, unavailable nodes, and strangers, we need to address two issues: how to pick relays and how to handle unavailable relays. Pretend for the moment that we know how to choose honest relays that—if available—would properly anonymize a querier.

To decrease the chance of a relay being unavailable, we can choose say j relays, ping them, and then use a random relay that responds. When an honest node responds to a ping, it commits to not voluntarily going down or dis-

connecting in the next $t_\Delta+2\epsilon$ time units. If no relays respond, the probably-anonymous query fails but this should happen with very low probability, say p_j, for sufficiently large j. This scheme handles nodes already unavailable at the start of a period, including nodes that have recently left, and nodes that do not crash (i.e., involuntarily become unavailable) during the query.

When a query's relay crashes, that query fails (e.g., (e) fails) so we must adjust p downwards from θp_d^4 (see Section 4) to $(1-p_j)\hat{\theta}p_d^4$ where $\hat{\theta}$ is the probability of a ping-responding relay being both honest and available for the duration of the query. Note that if dishonest relays are disproportionately available or non-crashing, then $\hat{\theta}$ is not simply θ times the probability of an honest node crashing during a query. For example, if 1% of system nodes are dishonest but only 90% of honest nodes are unavailable at any given time then 9.2% of the available nodes will be dishonest.

Our probably-anonymous query implementation will work for any method of secretly and randomly choosing a system node that has the property that knowledge of what node was picked does not (usually) reveal who picked it. One way to pick relays in the presence of churn is for each node to maintain a reasonably up-to-date membership roster and choose relays from it randomly. If the rosters are sufficiently up to date then all honest nodes should agree on a large core C of nodes whose membership has been stable recently. If we assume that honest nodes' rosters have at most 20% non-core nodes and conservatively regard choosing w from outside the core as revealing u's identity and let all dishonest nodes be in C, then we pick a relay w that will anonymize us with probability at least $\theta - \frac{2}{10}$.

A robust way to keep the rosters up to date is to use a gossip protocol where each node exchanges membership changes occasionally with other nodes chosen randomly from its roster list; the higher the churn rate, the more frequently gossip will need to be exchanged. If nodes are down much of the time, then a store-and-forward message system (e.g., email) may be needed for transmitting gossip. In order to maintain the required system invariant that the fraction of nodes that are honest is at least θ, one needs to use a mechanism to control entry (e.g., proof of solving an expensive problem). To prevent disrupters from lying about fake nodes joining or honest nodes leaving, signed joining and farewell messages can be used.

We think it should be possible to pick relay nodes without requiring each node maintain a full membership list; see the related work on peer sampling.

8 Extensions

8.1. Privacy

Adding our basic limiter implementation to a system may cause it to leak additional information: because our limiters work by forcing limited users (e.g., song providers) to tell anonymous probers who they are transacting with, anyone can figure out who a limited user is transacting with merely by probably-anonymously querying that user. To prevent this, users can use a different pseudonym for each partner, period pair. With this change, information leaks only to the relays that are directly talking to both parties. It should be possible to hide the transactors from the relays as well by using a series of relays instead of single relays as in Crowds [19]. The number of users a limited user is transacting with (for $k > 1$) can be hidden by padding his partner list with fake pseudonyms.

8.2 Disrupters

We have discussed the case where dishonest users wish to exceed the limits we are imposing. An alternative case involves *disrupters*, dishonest users that wish to disrupt the system by preventing as many transactions as possible (e.g., an attacker trying to prevent song trading). With our probably-anonymous queries, a single disrupter relay can block a transaction by returning "timeout" so even a small fraction of disrupters can substantially reduce the transaction rate of honest users. For example in the Musictella example where r=12, the 1% dishonest users could disrupt $d = 1-\theta^r = 1-.99^{12} = 11.4\%$ of the transactions (21.4% if receivers are also limited).

By increasing r and allowing the approval protocol to approve even with some incorrect answers from the probably-anonymous queries, it is possible to decrease the level of disruption. If we allow b incorrect answers then

$$\delta = |\Pi| \sum_{i=0}^{b} \binom{r}{i} p^i q^{r-i} \quad (1)$$

$$d = 1 - \sum_{i=0}^{b} \binom{r}{i} (1-\theta)^i \theta^{r-i} \quad (2)$$

where d is the fraction of transactions that are disrupted. (ℓ gets an extra transaction if at most b queries succeed and one of u's transactions is disrupted if more than b of its associated relays are disrupters.)

For this example, setting r to 15 and allowing 2 incorrect answers keeps 10δ at 0.1 and limits the disruption to 0.04% of transactions. Allowing some bad answers (including "timeout") also increases the system's resilience to communication failures, which can also block transactions.

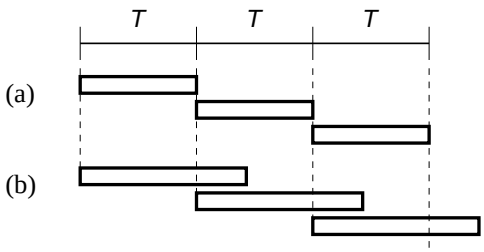

Figure 4. (a) non-overlapping periods of duration T. (b) overlapping periods that start every T time units.

Because there are so many possible relays and processing a single probe requires so little resources, a denial of service (DoS) attack against the relays is unlikely to be effective. Far more effective is simply requesting as many songs as possible, which consumes large amounts of resources. When song providers have their transaction rate limited below their natural transaction rate—whether by our limiters or a reputation system—the scarce resource being exhausted becomes allowed-transactions-per-hour instead of bandwidth. Rate limits thus can greatly amplify the effectiveness of any given song-request DoS attack. These attacks can be substantially mitigated, however, by additionally limiting nodes' ability to *request* songs by using a (second) transaction rate limiter.

8.3. Overlapping periods

We have described things assuming non-overlapping periods of size T. Because the approval protocol, which takes time $t_\Delta + t_r + 2\epsilon$ ($= 4t_d + t_r + 8\epsilon$), and any subsequent resulting transaction must run during a single period, this limits how small T can be. If this is a problem, consecutive overlapping periods of fixed length can be used, each of which starts T after the last period started (see Figure 4). This is equivalent to limiting the rate at which transactions can be started (e.g., k transactions can be started each T time units). Unlike with the non-overlapping case, the number of transactions a limited user has outstanding at any one time may exceed k in this case.

9 Related work

TCP's flow control and congestion control can be viewed as mechanisms to limit rates [12]. However, TCP's rate limiting applies to the flow between a pair of nodes, and is quite different from our rate limiters, which limit the *aggregate* rate of *transactions* between a node and all others.

To increase fairness, BitTorrent uses a tit-for-tat mechanism in an attempt to make each peer's download rate be proportional to their upload rate [4]. This mechanism works only between nodes simultaneously downloading the same content, does not limit the aggregate rate of a node, and is not suitable for systems where some nodes by design only consume or provide. Moreover, BitTorrent's mechanism can be circumvented using carefully modified clients [13, 17].

Like rate limiters, reputation systems are a way to prevent system misuse, but they are vulnerable to many attacks, including those from traitors [14]. Rate limiters can limit the gains obtained from such attacks so that it is not cost-effective for traitors to sacrifice their reputations.

Many reputation systems have been proposed; surveys of the field are beginning to appear [6, 14]. The reputation system we described in Section 2 was of necessity fairly simple. Among the proposed reputation systems, one of the more sophisticated is EigenTrust.

EigenTrust [9] assigns each node a global trust value that is intended to reflect that node's history of uploads. The assignment is based on each node's local experiences (how did X treat me?), a notion of transitive trust (weigh X's opinions based on how much we trust him), and a special set of trusted nodes. The trusted nodes are required for the distributed computation of global trust values—done via power iteration—to converge. In addition to its need for trusted nodes, which may be easy to take out or subvert, EigenTrust does not appear to handle churn, sometimes offline nodes, or disrupters. It seems particularly vulnerable to the traitor problem because it does not distinguish between "I've had a bad experience with this node" and "I've never talked to that node". We suspect EigenTrust has much greater overhead than our system, but it is hard to tell from the provided information.

Anonymity is an area of active research [1]. Chaum mixes [3] use public key cryptography and a set of computers called *mixes* to achieve anonymity. Onion routing [18] is basically a real-time variant of Chaum mixes. Crowds [19] achieves anonymity using a relaying mechanism rather than cryptography. Crowds gives a classification of degrees of anonymity (e.g., beyond suspicion, possible innocence, etc.), but it does not consider schemes that provide different degrees of anonymity probabilistically, such as our probably-anonymous queries. Such schemes allow for simpler and more efficient implementations than Crowds. Cashmere [23] and information slicing [10] are anonymization systems that explicitly consider churn. Several attacks on anonymization systems are described in [21].

The problem of randomly choosing a user in a (dynamic) group is called *peer sampling*, which can be achieved via gossiping [8]. If group membership is stable, peer sampling can be done with low latency and message overhead by using random walks, without knowledge of global membership [11].

10 Conclusion

In this paper, we have introduced transaction rate limiters, which impose limits on the rate at which users can perform transactions. Such limits have many uses, including restricting users to their fair share of resources, slowing spam to a trickle, and allowing traitors to be evicted before they can cause substantial damage. We showed how to implement transaction rate limiters in a robust manner using our probably-anonymous query implementation, which provides anonymity with at least a specified probability even against a collusion of a large fraction of the system's users.

Acknowledgments

We are thankful to the anonymous referees for providing useful feedback.

References

[1] Anonymity bibliography. http://freehaven.net/anonbib/full/date.html.

[2] R. Bazzi and G. Konjevod. On the establishment of distinct identities in overlay networks. *Distributed Computing*, 19(4):267–287, Mar. 2007.

[3] D. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–88, Feb. 1981.

[4] B. Cohen. Incentives build robustness in BitTorrent. In *Proceedings of the Workshop on Economics of Peer-to-Peer Systems (P2PEcon 2003)*, June 2003.

[5] J. Douceur. The sybil attack. In *Proceedings of the 1st International Peer To Peer Systems Workshop (IPTPS 2002)*, pages 251–260, Mar. 2002.

[6] K. Hoffman, D. Zage, and C. Nita-Rotaru. A survey of attack and defense techniques for reputation systems. Technical Report CSD–TR 07–013, Purdue University, May 2007.

[7] N. Hu, O. Spatscheck, J. Wang, and P. Steenkiste. Optimizing network performance in replicated hosting. In *Proceedings of the 10th International Workshop on Content Caching and Distribution (WCW 2005)*, pages 3–14, Sept. 2005.

[8] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems*, 25(3), Aug. 2007. Article No. 8.

[9] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina. The Eigentrust algorithm for reputation management in P2P networks. In *Proceedings of the 12th International World Wide Web conference (WWW 2003)*, pages 640–651, May 2003.

[10] S. Katti, J. Cohen, and D. Katabi. Information slicing: Anonymity using unreliable overlays. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2007)*, pages 43–56, Apr. 2007.

[11] V. King and J. Saia. Choosing a random peer. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing (PODC 2004)*, pages 125–130, July 2004.

[12] J. Kurose and K. Ross. *Computer Networking: A Top-Down Approach Featuring the Internet*. Addison-Wesley, Boston, 3rd edition, 2005.

[13] T. Locher, P. Moor, S. Schmid, and R. Wattenhofer. Free riding in BitTorrent is cheap. In *5th Workshop on Hot Topics in Networks (HotNets V)*, pages 85–90, Nov. 2006.

[14] S. Marti and H. Garcia-Molina. Taxonomy of trust: Categorizing P2P reputation systems. *Computer Networks*, 50(4):472–484, Mar. 2006.

[15] R. Merkle. Secure communications over insecure channels. *Communications of the ACM*, 21(4):294–299, April 1978.

[16] D. Mills. Network time synchronization research project. http://www.cis.udel.edu/∼mills/ntp.html.

[17] M. Piatek, T. Isdal, T. Anderson, A. Krishnamurthy, and A. Venkataramani. Do incentives build robustness in BitTorrent? In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2007)*, pages 1–14, Apr. 2007.

[18] M. G. Reed, P. F. Syverson, and D. M. Goldschlag. Anonymous connections and onion routing. *IEEE Journal on Selected Areas in Communications*, 16(4):482–494, May 1998.

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

[20] H. Rowaihy, W. Enck, P. McDaniel, and T. La Porta. Limiting Sybil attacks in structured peer-to-peer networks. In *Proceedings of the 26th Annual IEEE Conference on Computer Communications (INFOCOM 2007)*, pages 2596–2600, May 2007.

[21] M. K. Wright, M. Adler, B. N. Levine, and C. Shields. The predecessor attack: An analysis of a threat to anonymous communication systems. *ACM Transactions on Information and System Security*, 7(4):489–522, Nov. 2004.

[22] H. Yu, M. Kaminsky, P. B. Gibbons, and A. Flaxman. SybilGuard: defending against Sybil attacks via social networks. In *Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM 2006)*, pages 267–278, Sept. 2006.

[23] L. Zhuang, F. Zhou, B. Y. Zhao, and A. Rowstron. Cashmere: Resilient anonymous routing. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 2005)*, pages 301–314, Apr. 2005.