The VLDB Journal manuscript No.
(will be inserted by the editor)

Free Riding-Aware Forwarding in Content-Addressable Networks

Klemens Böhm[1], **Erik Buchmann**[2]

[1] Universität Karlsruhe (TH), Germany. e-mail: klemens.boehm@ipd.uni-karlsruhe.de
[2] Universität Magdeburg, Germany. e-mail: buchmann@iti.cs.uni-magdeburg.de

Received: date / Accepted: date

Abstract Research on P2P data structures has tacitly assumed that peers readily participate in the work, i.e., are cooperative. But such participation is voluntary, and free riding is the dominant strategy. This article describes a protocol that renders free riding unattractive, for one particular P2P data structure. The protocol is based on feedback that adjacent nodes exchange. This induces transitive logical networks of nodes that rule out uncooperative peers. The protocol uses proofs of work to deter free riding. To show that cooperative behavior dominates, we have come up with a cost model that quantifies the overall cost of peers, depending on their degree of cooperativeness and many other parameters. The cost model tells us that we can achieve a good discrimination against peers that are less cooperative, with moderate additional cost for cooperative peers. Extensive experiments confirm the validity of our approach.

Key words Peer-to-Peer distributed hashtables free riding incentives reputation

1 Introduction

Peer-to-Peer data structures (P2P data structures) address a core issue of data management research, namely administering huge sets of (key, value)-pairs. A P2P system consists of nodes, a.k.a. *peers*. Peers may issue queries, but they are also supposed to participate in the work, i.e., storage of data and evaluation of queries in the context of P2P data structures. By participating in the work, peers share the infrastructure costs (disk space, energy, network bandwidth etc.). P2P data structures do not have a centralized instance, a.k.a. *coordinator*, which monitors and controls the peers. So far, research on P2P data structures has tacitly assumed that peers readily participate in the work. But experience with P2P systems that are operational, notably file-sharing systems, indicates that this is not realistic [3]. Peers seek to minimize their costs. *Free riding* is the dominant behavior in the economic sense. Existing technology does not solve the problem for P2P data structures: Payment mechanisms [14] are vulnerable and have high infrastructure costs. Certified code or similar solutions [9] typically require a centralized certification instance, i.e., are not P2P. Proposals against free riding in mobile environments [5,21] are not applicable either. There, peers can observe the behavior of other peers in the same radio network cell and infer their degree of cooperativeness.

Our objective is the design of protocols for P2P data structures that render free riding unattractive. This article focuses on the evaluation of queries. It does so for *Content-Addressable Networks (CAN)* [24], a prominent P2P data structure. With the protocol envisioned, peers will only answer queries issued by *reliable nodes*[1], i.e., nodes that have correctly processed all recent incoming queries. New nodes or nodes with an unclear status must prove their reliability first, before benefiting from the system. At the same time, the costs of the protocol shall not be much higher than the ones of existing protocols. The design of such a protocol is difficult, for various reasons:

– In contrast to other P2P scenarios different from P2P data structures (cf. [8,11]), it is not only one peer, but a sequence of peers that processes a query. This makes the problem much more difficult, as this article will show. In our setting, peers cannot readily observe the behavior of other nodes: if a query remains unanswered, the issuer cannot say which other peer has not cooperated. From a slightly different perspective, peers hide their intentions. A statement like "Connection refused" will not be generated if a node does not participate in the work. Hence, standard recovery mechanisms for CAN [24] like expanded ring search or flooding will not work.
– The transition between 'reliable' and 'unreliable' is blurred: A peer that tends to be unreliable may process incoming queries from time to time, in particular if there is a direct advantage in sight. Further, the 'attitude' of a node may change at any time.

[1] In the context of this article, *reliable* and *cooperative* are synonyms.

- There is no centralized instance that provides information on the degree of reliability of a certain peer in an authoritative way.
- The network may be large. In general, a peer has not interacted before with a peer whose query it is supposed to process. However, when deciding if it should process the query, it can only rely on its own observations from the past, or on observations from other nodes that it deems reliable.
- Peers may change their identity at any time and sign up under a new identity.

Our three contributions are as follows: First, we propose a CAN protocol that differentiates between reliable and less reliable peers. In a nutshell, a peer generates positive or negative feedback on other peers in certain situations. It forwards the feedback to other peers, attached to messages that it would send out anyhow. Each node decides for itself if it 'trusts' the feedback from others, and uses the feedback it is aware of to estimate the reliability of other peers. A node processes a request from another node only if it deems the node reliable. Peers with low reliability must provide *proofs of work (ProW)* [4, 17] before they obtain a query result. The expectation is that peers decide to be reliable, instead of carrying out many ProW and ending up with higher overall costs. We explain why our setting requires ProW, in contrast to other scenarios [11].

Second, we develop a cost model for our protocol that predicts the expected overall costs of a peer, depending on its degree of reliability. Using this model, we show that (1) the protocol differentiates well between reliable and less reliable peers, i.e., less reliable peers end up with higher overall costs than reliable ones. Since the behavior of peers is not readily observable, negative feedback on fully reliable peers typically exists. Our protocol does not guarantee that entirely reliable nodes do not have to supply any ProW. However, we use our cost model to show the following: (2) additional costs for reliable peers, incurred by such ProW, are moderate, compared to other costs.

Third, we show the quality of our protocol by means of extensive experiments. They show that our cost model predicts the system behavior well and demonstrate that the protocol is operational in realistic and synthetic settings. We stress that we have a CAN implementation of our own that is fully operational, for large numbers of peers [7]. We have used this prototype for a lot of the experiments. Our experiments further show that the protocol is robust against changes of the environment, i.e., it differentiates well between cooperative and uncooperative nodes, and it is robust against most kinds of attacks. The protocol does so even if external characteristics like the global failure probability change significantly. Another important finding is that the protocol does not penalize neighbors of uncooperative nodes.

Summing up, this article is the first to investigate how to tackle free riding in P2P data structures where the behavior of peers is not readily observable, by means of an open protocol. The protocol proposed is truly Peer-to-Peer since it does not rely on any centralized instance.

The remainder of this article has the following structure: The next section reviews related work, Section 3 provides a brief recap of CAN. Section 4 demonstrates the necessity of countermeasures against free riding. Section 5 describes our new protocol. Section 6 provides a cost model for the protocol. Section 7 features a discussion, and Section 8 confirms our findings by extensive experiments. Section 9 deals with attacks on our protocol. Finally, Section 10 concludes.

2 Related Work

The database community has started investigating distributed data structures about a decade ago [19, 18]. The topic has attracted recent interest from other communities as well. This has resulted in many proposals, now referred to as *overlay networks*, *distributed hash tables*, *structured P2P networks*, etc. [16]. All suggestions for distributed data structures we are aware of tacitly assume that all nodes follow the protocol and do not do free riding.

The proposals mainly differ with regard to *contact-selection*, i.e., which are the peers a node can directly communicate with, and *routing-selection*, i.e., which contact does a node forward the current query to. High contact-selection flexibility together with high routing-selection flexibility is preferred; more recent proposals try to achieve both [16]. The topology of the key space is closely related to contact and routing selection. The key space of *CAN* [24] is a torus of d dimensions. Each peer maintains a contact list containing at least $2d$ immediate neighbors of its zone in the key space. Section 3 provides a detailed description of CAN. *CHORD* [28] organizes the data in a circular one-dimensional key space. Messages travel from peer to peer in one direction through the cycle, until the peer whose ID is closest to the key of the query has been found. Each peer keeps track of $log(n)$ other peers in the distance 2^{k-1} with $1 \leq k \leq log(n)$, where n is the number of peers in the system. *Pastry* [27] uses a Plaxton Mesh to store and locate its data. The forwarding algorithm is similar to the one of Chord. Each node maintains a routing table containing $log_{2^b}(n) \cdot (2^b - 1)$ contacts determined by common prefixes of the node-IDs. b is an exogenous parameter. *P-Grid* [1] is based on a virtual distributed search tree. Each node is addressed with a subtree-ID, which is the binary string representation of the path from the root to the peer on the leaf of the virtual tree. For each level of the tree, each node maintains a reference to another peer in the same subtree whose ID branches to a different subtree in the deeper levels. [16] features a detailed survey of the various approaches.

CAN differ from other approaches where contacts do not have to be direct neighbors in the key space. Consequently, the number of hops of a query in CAN tends to be larger. On the other hand, identifying free riders is a hard problem if contacts may be nodes somewhere in the key space. The reason is that the dissemination of reputation- or trust-related information becomes more difficult. This article limits the discussion to CAN and leaves the problem of feedback dissemination in the more general case for future work.

Free riding has received much attention in the context of P2P systems different from P2P data structures [23]. Some approaches use the concepts of trust and reputation to get rid of free riding. *Trust* is the subjective belief of a node in the honesty and the capabilities of another participant. [20] provides a comprehensive formal definition of trust as a computational concept. *Reputation* is derived from observations of the behavior of a participant in the past. Reputation systems collect, distribute and aggregate feedback [26]. Reputation systems in P2P networks have to work without trusted third parties, immutable peer identifiers and centralized services.

Various approaches have addressed these challenges. Based on the Generalized Prisoner's Dilemma, [11] addresses free riding with a game-theoretic concept. It shows that common game-theoretic strategies like Tit-for-Tat do not work well in P2P systems, and proposes a novel family of incentive techniques based on a reciprocative decision function. The function copes with untraceable defections, dynamic populations and asymmetric transactions by using private or shared histories. The approach tries to find a theoretically optimal solution and does not address realization issues.

From an algorithmic point of view, [2] was one of the first proposals of P2P-based reputation management. It is based on complaints, i.e., negative feedback. Given that $c(u,v)$ denotes the complaint from peer u about v with $u,v \in P$, the reputation value is $T(u) = |\{c(u,v)|v \in P\}| \times |\{c(v,u)|v \in P\}|$. High values of $T(u)$ indicate that u is not trustworthy. Each node stores the complaints it has generated in a global repository that all nodes can access. The repository is implemented as a P2P data structure (P-Grid). However, [2] does not discuss measures against insertion of spoof feedback. These seem to be indispensable, should the system become operational. Further, the peer that stores feedback about a certain node is a promising target for attacks and a single point of failure from the perspective of that node. With our work in turn, each peer runs a local repository for reliability information. This allows for a tight integration of message forwarding with reliability issues and is independent from global structures.

PeerTrust [29] incorporates feedback S as a numeric expression of the satisfaction earned by each transaction, the credibility C of the participating peers, and factors for the community- and transaction contexts (CF and TF). Let u denote the peer in question, $p(u,i)$ the other peer participating in the transaction i. Then the basic trust value is $T(u) = \sum_i S(u,i) \cdot C(p(u,i)) \cdot TF(u,i) + CF(u)$. A P-Grid instance stores the feedback items $S(u,i)$. This approach does not address all issues either. First, managing and distributing a comprehensive history of many feedback items is expensive. Second, the software plays the role of a trusted third party. The concept fails if the software does not guarantee that each transaction is rated one time, that only the participating nodes have permission to write feedback to the P-Grid repository, and that each peer can access it. Third, the vulnerability of the P2P repository against directed attacks against peers responsible for certain feedback items remains, analogously to [2].

Similar approaches exist, e.g., EigenTrust [13]. However, those vulnerabilities of global P2P repositories call for local interactions between transaction partners instead. Here, the feedback about a certain node is distributed among its transaction partners. They may assess the same peer differently. [22] describes sharing of reputation information in such a decentralized way. A node describes every other node with a rating coefficient, i.e., a numeric value. Nodes share the coefficients after each transaction. A node updates its coefficients by adding the new value weighted by the coefficient of the sender. However, the approach does not pursue a tight coupling of reputation management and query processing and is less specific than ours.

Another concept to rule out uncooperative behavior uses micropayments [14]. However, infrastructure costs with micropayments may simply be too high in a setting such as ours with many small transactions. More recent approaches like PPay [30] try to reduce the infrastructure costs by using floating, self-managing coins, i.e., the load is transferred from the broker to the peers. But in general, payment schemes based on artifical coins may come along with inflation or deflation. Furthermore, they typically require a central bank in order to prevent from fraud coins.

Public-key cryptography can help to ban free riders. There are two approaches [10]: The quorum-based model assumes that the majority provides true information. The Web-of-Trust model builds trust graphs from the peers trusted by the issuer, until the peer in question is reached. However, both approaches are expensive: the first one requires the invocation of many nodes, the second one is similar to the traveling salesman problem. Other approaches use certified code or similar technologies [9] to prevent users from behaving at will. But this requires a central instance, and would not be in line with P2P. Dealing with free riders in mobile networks is simpler than in our setting, and existing solutions are not applicable either [5,21]: A core difference is that nodes can eavesdrop messages to and from nodes in the same radio network cell.

[6] features a preliminary version of our protocol that lacks important features described here. There, only positive feedback is used, but not negative feedback. For this reason, a peer can only recognize that another peer has turned uncooperative when the respective feedback times out. While negative feedback is indispensable, it significantly adds to the complexity of the protocol. Further, the evaluation of the protocol in this current article is much broader. It includes a formal analysis and an experimental demonstration that our model is a faithful description of reality.

3 Content-Addressable Networks

Content-Addressable Networks (CAN) [24] may serve as the basis for a broad variety of applications in the realm of the WWW, Semantic Web or elsewhere; see [15] for a comprehensive list. They address a core issue of data management

research, namely administering huge sets of (key, value)-pairs under high query and update rates.

A CAN is a distributed system that consists of many nodes (peers). Nodes typically are PC or workstations, operated and maintained by different persons or organizations. A peer benefits from the CAN by issuing queries and obtaining query results. On the other hand, it is supposed to participate in the administration of the data and in the evaluation of queries. Each CAN node is responsible for a part of the key space, its *zone*. The key space is a n-dimensional torus of Cartesian coordinates in the unit space, and is independent from the physical network topology.

A query is a point in the key space. We also speak of *query points*. The query result is the value corresponding to the query point. In addition to its zone, a node knows all neighbor nodes, i.e., nodes whose zones are adjacent to its zone, and stores them in a *contact list*.

Example 1 The key space of the CAN in Figure 1 is two-dimensional. The Node P_1 is responsible for Zone ([0.5; 0.5), [0.625; 0.75)) of the key space, i.e., it knows all (key, value)-pairs where key $\in$ ([0.5; 0.5), [0.625; 0.75)). The neighbors in the contact list of Node P_1 are Nodes P_2, P_7, P_4, P_9, P_3 and P_6. Node P_2 is a neighbor of Node P_8.

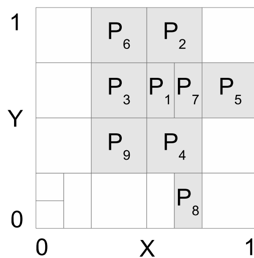

Fig. 1 Two-dimensional CAN.

The partitioning of the key space results from the CAN construction protocol. A peer which wants to join the CAN finds a random node that is already in the CAN. That node splits its zone, keeping one half and reassigning the other half to the new node. Finally, the two nodes inform all neighbors about the new zone assignment.

Given this key space partitioning, query processing is a variant of *greedy forward routing* based on the Chessboard distance. The original proposal from [24] uses the Manhattan distance. But our feedback dissemination mechanism depends on the fact that two peers with adjacent zones have some neighbors in common (see Section 5). Zones typically have different sizes. $2d$ is a lower bound on the number of neighbors in a CAN using the Manhattan distance. $3^d - 1$ is the lower bound for the Chessboard distance. A larger number of neighbors induces a higher maintenance overhead, but decreases the average path length in the CAN. [24] features a more detailed discussion of the various tradeoffs involved here.

A node that has issued a query first checks if it can answer the query. This is the case if the query point falls into its zone. Otherwise, it forwards the query to that neighbor in its contact list whose distance to the query point is minimal. The procedure recurs until the query arrives at the node that can answer it, the *target node*. The target node then sends the result to the issuer.

4 How Bad is Free Riding in CAN, after all?

Obviously, any mechanism to deal with uncooperative behavior comes with additional computational overhead. On the other hand, when looking at well-known coordinator-free P2P systems in the wild, e.g., Kazaa or gnutella, it seems that a P2P system can function in spite of free riders. But there is a big difference between this kind of P2P systems and P2P data structures: There, peers *flood* the network with a request, i.e., a peer sends a replica of a query message to each of its contacts (except for the one which has sent the query to the peer). In contrast, P2P data structures use *greedy forwarding*, i.e., a message goes from peer to peer. Obviously, resource consumption with this scheme is much less. Further, any peer can access any (key, value)-pair. On the other hand, P2P data structures are vulnerable when it comes to loss of messages.

In a CAN consisting of $N = 10,000$ peers with $d = 4$ dimensions, a query is forwarded $l = d/4 \cdot N^{1/d} = 10$ times on average (cf. [24]). Suppose that the CAN contains $u = 500$ peers which do not forward any incoming query message. Then the probability of obtaining a query result is only $(1 - u/N)^l \approx 60\%$. Furthermore, the rate of answered queries decreases exponentially when the failure probability goes up.

An experiment illustrates the problem further. We have run a CAN consisting of $N =$ 100,000 peers with a four-dimensional key space. We varied the number of free riders u from 0 to 50% and the private failure rate q, i.e., the rate at which free riders do not forward or answer messages, from 0 to 100%. Figure 2 shows the result of the experiment. The z-axis is the rate of queries answered a. The experiment shows that in settings without countermeasures against uncooperative nodes even a small number of free riders or a small failure rate reduces the number of queries answered significantly.

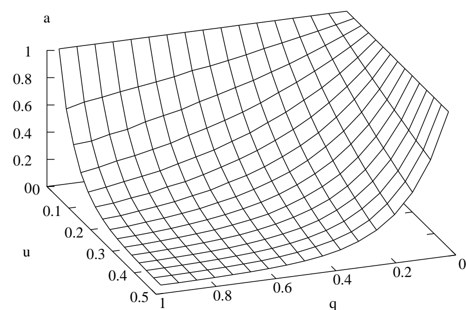

Fig. 2 Rate of answers obtained at different rates of uncooperative peers and private failure rates.

Now suppose a peer would repeat an unanswered query after a certain period of time, and the P2P data structure has

many replicas or uses m-of-n data coding, i.e., zones assigned to free riders would not result in any loss of data. Then the presence of free riders would 'only' incur additional network expenses and an increased latency. Now we ask: Where is the break-even, compared to a protocol that excludes any free riding at the price of additional network overhead?

To answer this question, we first determine the average number of message forwards $\overline{f}$ in the presence of free riders, i.e., how many peers forward a message on average given that each peer drops the message with probability p? Let $l = d/4 \cdot N^{1/d}$ be the average length of a forwarding path under the assumption that peers do not drop messages. $u/N = p$ denotes the global failure probability, i.e., the probability of a peer to not process a query message. Obviously, in a CAN without free riders it is the case that $\overline{f} = l$. In a CAN where the global failure probability is $p = 1$, the message is forwarded from the issuer to its neighbor and dropped right away, so $\overline{f} = 1$. In general, the average number of message forwards $\overline{f}$ is the sum of the probabilities that the message is dropped after $1 \cdots l$ forwards or answered after l forwards, multiplied with the respective number of forwarders:

$$\overline{f} = \Big(\sum_{i=1}^{l} (1-p)^{i-1} \cdot p \cdot i \Big) + (1-p)^l \cdot l \qquad (1)$$

Suppose that each peer repeats an unanswered query up to t times. Given the costs c_m of transmitting a query message from one peer to another, the average cost $\overline{c_n}$ comes from the cost of each try multiplied with the probability to obtain an answer after $1 \cdots t$ tries:

$$\overline{c_n} = \overline{f} \cdot c_m \cdot \sum_{j=1}^{t} \big(1 - (1-p)^l\big)^{j-1} \cdot (1-p)^l \qquad (2)$$

Now consider a protocol that rules out free riders at the cost c_a of feedback on peers attached to each outgoing message, such as our protocol[2]. We estimate the average costs $\overline{c_r}$ as follows:

$$\overline{c_r} = l \cdot (c_m + c_a) \qquad (3)$$

Let us now return to our original question: Where is the break-even of a protocol that incorporates mechanisms against free riding, given a realistic setting?

Experiences from P2P systems that are operational such as gnutella indicate a rate of free riders of up to 70% [3]. However, we believe that P2P data structures will attract another target group. We therefore assume a rate of 10% in the calculation that follows. (If we used those 70%, our protocol would look much better.)

Now consider an application of P2P data structures where the keys are URLs. The costs of transferring messages are assumed to be the number of bytes. To obtain a realistic distribution of these numbers for URLs, we have implemented an experimental web crawler. It runs a modified random walk algorithm over the WWW.[3] Table 1 shows some statistics. Next to the URL itself, a query message contains some protocol overhead, e.g., information about the issuer and some flags. In addition, a TCP/IP frame wraps each message, and SYN- and ACK-messages initiate each TCP/IP-connection. We estimate (very) generously $c_m = 200$ bytes.

Section 6 will show that forwarding 10 feedback items is sufficient in order to identify and exclude the vast majority of free riders. A feedback item is a data object consisting of a Peer-ID, a timestamp and some flags (cf. Section 5). Thus we set $c_a = 100$ bytes.

Number of pages crawled	7,646,238
Minimum length of URLs crawled	12.0
Maximum length of URLs crawled	255.0
Average length of URLs crawled	61.1
Standard deviation of the length	18.7

Table 1 Some statistics from our experimental web crawler.

Figure 3 now graphs the overall costs (y-axis) of a CAN protocol without measures against free riding and the ones of a protocol incorporating such mechanisms. The number of peers ranges from 1,000 to 500,000 (x-axis), d =4, and the number of retries is not limited (in the context of the first protocol). As expected, a larger number of peers increases the costs due to longer paths and higher failure probabilities. As a result, even when the rate of 10% uncooperative peers is small, the free riding-aware forwarding protocol outperforms the standard protocol. Obviously, one could construct scenarios where peers fare better without our protocol. But the settings behind these scenarios are unrealistic or impractical. For example, the break-even is below 2.9 % uncooperative nodes in a CAN consisting of 500,000 nodes, or it is below 2500 nodes in a setting with 10% uncooperative nodes.

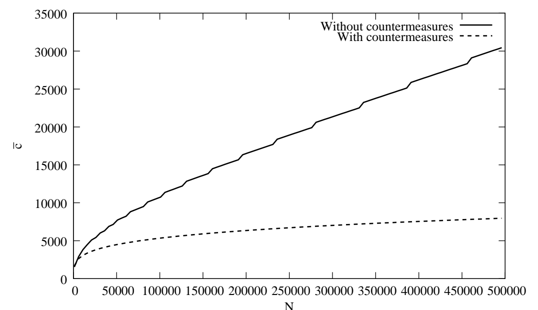

Fig. 3 Network traffic in bytes for operating with or without our countermeasures in a CAN with 10% uncooperative nodes.

[2] We assume that ProW are requested from free riders only.

[3] The algorithm caches URLs visited. When a web server returns an error, the algorithm restarts the random walk from a randomly selected URL from the cache.

An analysis of the delay for obtaining query answers yields similar results. The only difference is that attaching additional information to messages practically does not increase the latency. Even when looking at the delay in isolation already and leaving aside the costs, countermeasures against free riders are always advantageous.

5 A CAN Protocol that Differentiates between Cooperative and Uncooperative Peers

This section describes our new CAN protocol. We start with a description of our various assumptions, followed by one of the new data structures. Then we give an overview of our protocol, followed by the protocol itself.

5.1 Assumptions

Our reliability-aware CAN protocol depends on certain characteristics of the nodes and of the applications using the CAN, as described next. We will address the impact of these assumptions in Sections 7 and 8.

Application profile with frequent queries and small query results. This article focuses on an application profile with the following characteristics: Peers remain connected to the network for a long time. They issue queries frequently and regularly. Query results are typically small, thus their delivery is not much more expensive (in terms of infrastructure costs) than query routing. Results are needed in time, so it is infeasible to batch queries and issue them at once. It is acceptable if some (very few) queries remain unanswered. Examples are object lookup systems, annotation services, push services etc. Thus, we strive for lightweight mechanisms that must cope with a high rate of small parallel queries.

Equal private costs. A general problem is that the costs of a peer (CPU, network bandwidth, memory, etc.) are *private information*. For example, a peer connected with a dial-up modem wants to save bandwidth, as opposed to one with a leased line, which might be better off carrying out (CPU-intensive) ProW instead of processing queries. In general, it is not feasible to observe these preferences. This article assumes equal private costs for all peers. Further, a node itself is responsible for keeping the rate of system failures low. Our protocol does not (and cannot) 'feel sorry' for peers that are cooperative, but run in an unstable environment.

Messages are not tampered with during forwarding. We assume that only the issuer of a piece of information can have falsified a message. For example, a peer may create false feedback. But it is unable to perform a man-in-the-middle attack, e.g., to intercept a response message and claim to be the peer who has provided the query result. In the presence of cryptographic signatures and the unlimited connectivity of the Internet, this is realistic.

No uncooperative behavior at application level. This article leaves aside adverse behavior on the application level. For instance, a node might try to prevent other nodes from obtaining access to a certain (key, value)-pair and might attempt a DoS attack on the node responsible for the pair. When looking at the storage level in isolation, such an attack consumes resources, but does not provide any benefit to the node that initiates it. While uncooperative behavior at the application level is an important problem, it is beyond the scope of this article. The problem of free riding at the storage level has to be solved first.

Verifiability of query results. The issuer of a query must be able to verify the correctness of the result. Otherwise, a node could send back a spoof query result and save the cost for data storage. Verification of query results can take place by exploiting replication or application-specific characteristics of the data values managed (cf. Section 9).

5.2 Data Structures and Message Components

With our protocol, each node decides individually if it deems another peer reliable, based on observations from the past. We refer to such observations as *feedback*. Feedback is time-stamped with the creation date and refers to one node, the *feedback subject*. Feedback items can be positive or negative. They always have the same weight. Each node manages a private *feedback repository* to keep up to s feedback items about each of its neighbors. We refer to the number of positive feedback items in the reputation repository of Node P_1, associated with subject P_2, as the *reliability coefficient of P_2 by P_1*. A coefficient below a threshold value t means that P_1 deems P_2 unreliable.

To keep track of queries it has forwarded in the past, a node maintains a *query log*. It contains the ID of the neighbor the node has forwarded the query to, and the query point. The query log is purged from time to time. A so-called *feedback notification* informs a peer about the success or failure of a query it has forwarded recently. Finally, the data structures from the conventional CAN are present as well, notably the contact list and the structures storing the (key, value)-pairs.

5.3 Overview

The principle of our protocol is that nodes collaborate with reliable neighbors only. If Node P_1 sends a query to its Neighbor P_2, P_2 will process the message in an outright way only if it deems P_1 reliable. Thus, it is in the interest of a node that its reliability coefficients by its neighbors are high. Our protocol offers three alternatives for a node P_1 to become reliable in the eye of a certain neighbor P_2, i.e., obtain a reliability coefficient $\geq t$:

R1 P_1 forwards or answers a query, and P_2 receives a notification about this.

R2 P_1 answers a proof of work request from P_2. (Will be explained right away.)

R3 Neighbors of P_1 share their feedback, and the number of positive feedback items on P_1 in the repository of P_2 is at least t.

In addition, there are three ways how P_1 can become unreliable in the eyes of P_2:

R4 P_1 does not forward or answer a query, and P_2 receives a notification about this.

R5 P_1 forwards a query, but another peer does not forward or answer it later on.

R6 Neighbors of P_1 share their feedback, and the number of positive feedback items on P_1 in the repository of P_2 is below t.

Example 2 The repository of P_1 contains $s = 10$ feedback items on P_2. P_1 has generated three of them itself. Two of them are positive. P_1 has generated them because other CAN nodes have answered queries issued by P_1, and P_1 had forwarded these queries to P_2. The third feedback item is negative. From the remaining 7 feedback items obtained from other nodes, 6 are positive, and one is negative. Thus, the reliability coefficient of P_2 by P_1 is 8. Assuming that $t \leq 8$, P_1 deems P_2 reliable.

Now think of a Node P_3 that obtains a query forwarded by P_2, initially issued by P_1. P_3 estimates the reliability of P_2. If the reliability coefficient of P_2 by P_3 is lower than t, P_3 asks P_2 for a ProW before processing the message. If the reliability coefficient is t or more, or if the ProW response arrives in time, P_3 processes the query.

Feedback is not always generated when **R1**, **R2**, **R4**, or **R5** occur, but only with a certain probability[4]. This is in order to indirectly assign different weights to the different kinds of feedback. For instance, if we think that **R1** is twice as important as **R5**, the probability assigned to **R1** is twice the one of **R5**. Implementing weights with probabilities is easier than with counters, and peers save bookkeeping efforts.

P_2 can observe **R2** only. With **R1**, **R4**, and **R5**, an incoming feedback notification triggers the generation of feedback. It is the issuer of the query that generates such a notification. For positive feedback, this happens after the arrival of the query result. If a result does not arrive within a certain period of time, the issuer sends out a negative feedback notification.

The rationale behind **R5** is that the issuer of the query does not know which peer has defected. **R5** ensures that the unreliable peer obtains negative feedback by penalizing all forwarders. This is of course quite undifferentiated. But the expectation is that unreliable peers end up with much more negative feedback on average (cf. Section 6). Furthermore, **R5** motivates the peers to search for reliable paths, e.g., to bypass (otherwise reliable) peers which tend to forward to unreliable nodes.

Feedback notifications work as follows: Every time a peer issues or forwards a query, it logs the addressee of the message. Now assume that the query result arrives. The issuer of the query reads the peer it has forwarded the query to from the log, generates positive feedback on it, and puts this feedback into its feedback repository. It then informs that peer with a feedback notification. The procedure recurs until each peer in the forwarding chain has generated positive feedback on the next forwarder. The processing scheme for negative feedback is analogous.

[4] Here, each peer is initialized with the same set of probabilities which do not change over time. The problem of variable, individual probabilities is left open for further research.

Whenever P_1 obtains a message from Neighbor P_2 that it does not deem reliable, P_1 requests a *proof of work (ProW)* from P_2. A ProW is a task that is easy to formulate, and the solution is easy to verify, but solving it requires a lot of resources [4, 17]. Having obtained a ProW from P_2, P_1 generates a number of positive feedback items on P_2. ProW are a waste of resources, when looked at in isolation. To decrease the number of ProW requested from reliable nodes, adjacent peers share feedback (**R3**, **R6**). In order to save resources, feedback is piggybacked to messages a peer sends out anyhow. In our example, P_1 forwards feedback to the nodes that are neighbors of P_2. A new feedback that is recent, be it incoming, be it generated by the node itself, replaces the oldest item in the repository. This is in order to react to changes in the behavior of nodes. To make dissemination of spoof feedback more difficult, a node accepts feedback only from neighbors that it deems reliable. A detailed discussion on spoof feedback and other potential attacks is provided in Section 9.

At first sight, since new nodes are likely candidates for ProW, it might seem that ProW are disincentives to join the network. But there is no alternative to 'entrance fees' for newcomers in settings where peers can change their identity at little cost. This is because a P2P system does not contain a central instance that authenticates and monitors the users. With our protocol, a user could erase his reputation by leaving the network and joining under another IP address and node ID. Giving advances to newcomers is likely to induce them to consume the advances, then leave the network and join under a new identity. Related work in game theory confirms that the strategy with the highest payoff employs entry fees. For example, [12] formally proves based on the prisoners dilemma that no strategy can do better than one that punishes newcomers.

There are ways to ease the join process for newcomers. Consider again the CAN construction process sketched in Section 3. A new peer receives one half of the zone and information about the neighbors from a node that is already part of the system. An obvious extension to this mechanism is to pass on feedback information on the neighbors as well. An assumption behind this is that a peer that helps a newcomer is reliable. Another extension to the CAN protocol could deploy public-key cryptography to allow peers to keep its reputation while being logged out.

One might wonder why the entrance fee cannot be 'regular' work, instead of a ProW. 'regular' means that the nodes wait until a neighbor node asks them to perform useful work, in our case the forwarding and answering of queries. Unlike other P2P settings, query processing in P2P data structures requires the cooperation of a sequence of peers. If only one peer in a forwarding chain defects, the query is not processed successfully. All peers in the chain would obtain negative

feedback. In their own self-interest, nodes must be able to find out about the degree of cooperativeness of a peer that is not known as cooperative to any other reliable node. Furthermore, this must go on fast. It is not feasible to wait for the peer in question having performed enough regular work.

So far, ProW have been artificial problems like finding prime factors of large numbers. However, ProW that perform useful work from the perspective of the application are conceivable as well. The only prerequisite is that the problem is hard to compute, but it is easy to formulate, and its solution is easy to verify. For example, if a CAN stores data crawled from the web, a ProW might be a crawl of a certain WWW domain. The peer which wants to verify the ProW checks whether a few randomly selected pages from the domain are part of the crawl. It might be possible that the validation of such a ProW is successful, even though the peer in question has not crawled the entire domain. But the probability of this can be kept arbitrarily small.

It remains to be discussed why a node should carry out a ProW in the context of queries issued by other nodes. When assuming that nodes issue queries at a steady rate, the node will soon issue a query itself. If its reliability coefficient is below t, it will have to carry out a ProW anyhow (cf. Subsection 5.1). Furthermore, a ProW delays query processing. If a node refuses ProW until they relate to its own queries, it is the processing of exactly these queries that is delayed.

5.4 Protocol

We now describe the various methods that implement our protocol. To ease the presentation, we assume that a key is only queried for once. Of course, our implementation can deal with the general case, i.e., keys being queried for more than once.

```
1  query(Point x) {
2      // forward the query message
3      handleQuery(x, this, this);
4
5      Result r := waitForAnswer(timeout);
6      if (query result returned in time) {
7          handleFeedbackNotification(x,
8              answer obtained, this);
9          return r to application;
10     } else {
11         handleFeedbackNotification(x,
12             no answer obtained, this);
13     }
14 }
```

Fig. 4 Method *query*

Method query: If a peer wants to obtain the value corresponding to a certain query point, it will invoke Method *query* (Figure 4) with the query point as parameter. Method *query* invokes *handleQuery*, which is described below, and waits for an answer. If it arrives in time, *query* initiates the generation of positive feedback and of a positive notification, by calling *handleFeedbackNotification*. In addition, the method returns the query result to the application. If the query result does not arrive in time, *handleFeedbackNotification* is invoked with a negative parameter value.

```
1  handleFeedbackNotification(Point x,
2      NotificationType n, LastForwarder f) {
3
4      // is the last forwarder of the notification reliable?
5      if (f ≠ this ∧ f.reliabilityCoefficient < t) {
6          stop processing the feedback notification;
7      }
8
9      // get the peer this node has forwarded the query to
10     Addressee a := this.queryLog.get(x);
11
12     // generate feedback and forward the notification
13     generateFeedback(a, n);
14     send(a, NotificationMessage(x, n));
15 }
```

Fig. 5 Method *handleFeedbackNotification*

Method handleFeedbackNotification: This method (shown in Figure 5) is invoked with every observation of work performed or not performed. This may be a returned query result as well as the arrival of a feedback notification. Method *handleFeedbackNotification* is invoked with the query point, the notification type which can be positive or negative, and the last forwarder of the notification (or the one that has generated it). *handleFeedbackNotification* first checks the reliability of the last forwarder. If it is not the peer itself and is not reliable, the feedback notification is classified as spoof and is ignored. Otherwise, *handleFeedbackNotification* reads the addressee of the query from the log, and generates feedback with the respective peer as feedback subject. At last, *handleFeedbackNotification* sends a feedback notification to that peer, and the process recurs. An obvious optimization which is also part of our implementation is to ship feedback notifications piggybacked to regular messages, instead of separate messages.

Method generateFeedback: This method creates feedback items and stores them in the local feedback repository. The first parameter of Method *generateFeedback* is the feedback subject. The second one specifies the reason why feedback is created, i.e., *NotificationType* is an enumeration type of the following values: *query result obtained*, *no query result obtained*, *forward*, *no forward*, *correct ProW result delivered*. Given this second parameter value, Method *generateFeedback* generates a feedback item with a certain probability (cf. Table 2).

Method handleQuery: This method (Figure 6) answers and forwards queries to reliable peers and generates feedback attachments. The parameters of *handleQuery* are the query

```
1  handleQuery(Point x, LastForwarder f, Issuer i) {
2      // is the last forwarder reliable?
3      if (f ≠ this ∧ f.reliabilityCoefficient < t) {
4          requestProW(f);
5          waitForProWAnswer(timeout);
6          if (ProW answer returned in time)
7              generateFeedback(f, ProW obtained);
8          else
9              stop processing the query;
10     }
11
12     // answer from local zone?
13     if (x is located in this.zone) return value(x) to i;
14
15     // forward query
16     CandidatePeers C := {p | dist(p,x) < dist(this,x)
17         ∧ p ∈ this.contactList};
18
19     // determine reliable addressee
20     ReliablePeers R := {p | p.reliabilityCoefficient ≥ t
21         ∧ p ∈ C};
22     if (R ≠ ∅) {
23         sort R by dist(p ∈ R, x);
24         Addressee a := getFirstElement(C);
25     } else {
26         sort C by dist(p ∈ C, x);
27         forall (p ∈ C) {
28             requestProW(p);
29             waitForProWAnswer(timeout);
30             if (ProW answer returned in time) {
31                 generateFeedback(p, ProW obtained);
32                 Addressee a := p;
33                 break;
34             }
35         }
36     }
37
38     // generate feedback attachment
39     FbAttachment F := {f | isNeighbor(a, f.subject)
40         ∧ f ∈ this.repository };
41
42     // forward messages
43     send(a, QueryMessage(x, i, F));
44     this.queryLog.add(x, a);
45 }
```

Fig. 6 Method *handleQuery*

point, the last forwarder of the query (or the one that has issued it) and the issuer of the query. First, the method checks the reliability of the last forwarder. If it is below t, the peer is asked for a ProW. The query is ignored if there is no ProW result within the timeout period. *handleQuery* then checks if the query point is in its zone. If so, it returns the answer (or at least the information that there is no value for this query point) to the issuer of the query. Otherwise, the method identifies a peer to forward the query to. It is the neighbor node with the smallest distance to the query point that is reliable. If there is no such reliable neighbor, all neighbors are ordered by their distance to the query point. *handleQuery* now requests a ProW from each of them, one by one. As soon as one peer returns the ProW, it becomes the addressee, and positive feedback for the ProW is generated. Having identified the addressee, the method determines the feedback items to be piggybacked to the message. This is done by selecting all feedback items whose subjects have a zone adjacent to the one of the addressee. (Our implementation features some optimizations that ensure that no feedback item is sent to the same node twice.) At last, the query is forwarded, and the addressee is written to the query log.

Example 3 In Figure 1, assume that P_1 is about to forward a query to P_7. Then it would attach feedback on P_2 and on P_4 to the message.

Method *handleQuery* makes sure that the peer hands the query to a peer that it deems reliable. This is an important design decision – **R5** in Subection 5.3 states that a peer that forwards a query to an unreliable node is treated as if it was unreliable itself.

6 Formal Analysis

Parameter	Abbr.	Default
number of nodes per dimension	n	10
dimensionality of the key space	d	4
degree of replication	r	
total number of nodes	N	
size of feedback repository per neighbor	s	10
global failure probability for all nodes	p	
private failure rate	q	
threshold for reliability	t	
probability that one feedback item for forwarding is generated	$q_{forward}$	0.2
probability that one feedback item for query answering is generated	q_{answer}	0.5
relationship between positive and negative feedback items	q_{pn}	2
number of feedback items generated for ProW	q_{ProW}	1
cost of forwarding a message	$c_{forward}$	2
cost of answering a message	c_{answer}	5
cost of a ProW	c_{ProW}	100

Table 2 List of parameters.

Our analysis is based on various assumptions; we will address the impacts of our assumptions in Section 8: The CAN grid is completely regular, with n zones per dimension. The number of zones is n^d, where d stands for the dimensionality of the key space. Both the query points and the nodes issuing queries are uniformly and independently distributed in the key space. Query processing takes place in rounds. Each node issues one query per round. If a query remains unanswered, the issuer does not repeat it. The total number of nodes is

$N = r \cdot n^d$, where r is the degree of replication. The parameter r is identical for all zones. If d and r are sufficiently large, a node will always find a reliable neighbor closer to the target point than itself. Consequently, we can neglect situations where greedy forwarding fails because all neighbors in one direction are unreliable. Table 2 is a list of the parameters used. Their default values will be discussed in Section 7. We assume that the system is in steady state. Assuming steady state is not a restriction, see [13] for a description how a P2P system reaches such a state. In addition, utility equals cost, i.e., peers are risk-neutral.

In what follows, we use P_i to refer to a peer in the CAN, and $x_i = (x_{i1}, ..., x_{id})$ to refer to an arbitrary point in the zone of P_i. As a first step, we seek a formula for the number of hops of a message from one point to another one. Given two points $x_1 = (x_{11}, ..., x_{1d})$ and $x_2 = (x_{21}, ..., x_{2d})$; x_1, $x_2 \in [0;1]^d$ on a d-torus, their *cell distance in Dimension* i is as follows:

$$dist_i^n = \min(|\lfloor x_{1i} \cdot n \rfloor - \lfloor x_{2i} \cdot n \rfloor|, \\ 1 + \lfloor \min(x_{1i}, x_{2i}) \cdot n \rfloor + \lfloor n - \max(x_{1i}, x_{2i}) \cdot n \rfloor) \quad (4)$$

Example 4 In Figure 7, $n = 8$ and $d = 2$. The cell distance of a point in Cell P_1 and one in Cell P_5 in Dimension x is 4, it is 3 in Dimension y.

If the number of cells per dimension n becomes larger, the cell distance per dimension of two points becomes larger as well, irrespective of the fact that the positions of the points have not changed. Given the cell distances per dimension, the *cell distance* of two points is the L_∞-distance (*Chessboard distance*). In what follows, we use the notation $dist^{n,d} = \max_{i=1}^d (dist_i^n(x_1, x_2))$.

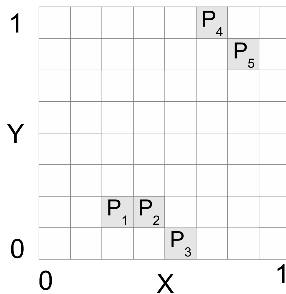

Fig. 7 Cell distance in a regular grid.

Example 5 The grey cells in Figure 7 represent one shortest path (out of several ones) from P_1 to P_5. Its length is 4. Now consider a message forwarded along this path. We say that P_2 is the *first forwarder of the query*, P_3 the *second forwarder of the query* etc. In our terminology, P_1 and P_5 are *issuer* and *answerer*.

What is the average number of message hops per query? Our assumptions are that nodes issue queries with the same frequency, and that the query points are uniformly and independently distributed. Thus,

$$avgdist^{n,d} = \iint_{x_1, x_2 \in [0;1]^d} dist^{n,d}(x_1, x_2) dx_1 dx_2 \quad (5)$$

The double integral calculates the distances between each possible source and destination point in d dimensions. Note that we obtain the average without dividing by the lengths of the intervals, because it is 1 in the unit space $[0;1]^d$. Formula 5 reflects the situation where no node along the path defects. The number of queries issued per round is N, the number of queries the node is expected to forward per round without defection and failures is $avgdist^{n,d}$.

Defection and Failures. Let p be the *global failure probability* over all nodes, including both system failures as well as adverse behavior resulting in queries that are not forwarded or answered. E.g., in a CAN consisting of 99 fully cooperative peers, one fully uncooperative peer that is unknown, and in the absence of system failures, $p = 0.01$. Our assumption that the CAN is large and in steady state implies that p is constant and does not change over time. In what follows, we refer to a node with failure probability p as cooperative, irrespective of the nature of the failures.

Each peer is supposed to forward queries issued by other nodes. The issuer delivers the query to the first forwarder with certainty. The first forwarder forwards it to the next one with probability $1-p$. The scheme recurs until the query reaches a peer that answers it, again with probability $1-p$. We assume that $N >> avgdist^{n,d}$, so these probabilities per hop are independent from each other. Thus, a query will be transmitted over a distance of δ hops and answered with probability $(1-p)^{\delta-1} \cdot (1-p)$. The expected number of forwarders of a query is the sum of the probabilities of being forwarded by a number of 1, 2, ... $dist^{n,d}(x_1, x_2) - 1$ forwarders. Given this, the average number of forwards each peer is expected to do per round is as follows:

$$h_{forward}^{n,d} = \iint_{x_1, x_2 \in [0;1]^d} \sum_{i=1}^{dist^{n,d}(x_1,x_2)-1} (1-p)^{i-1} dx_1 dx_2 \quad (6)$$

Clearly, in the presence of failures $p > 0$ and therefore is $h_{forward}^{n,d} < avgdist^{n,d}$. The number of queries a node is expected to answer per round depends on p as follows:

$$f_8^{aux}(p, \delta) = \begin{cases} 1 & \text{if } \delta = 0 \\ (1-p)^{\delta-1} & \text{if } \delta > 0 \end{cases} \quad (7)$$

$$h_{answer}^{n,d} = \iint_{x_1, x_2 \in [0;1]^d} f_8^{aux}(p, dist^{n,d}(x_1, x_2)) dx_1 dx_2 \quad (8)$$

Equation 7 differentiates between queries that are issued by other nodes and queries the issuer can answer itself, i.e., $dist^{n,d}(x_1, x_2) = 0$.

How much feedback should a cooperative node expect per round? A cooperative node obtains both positive and negative feedback. Consider the evaluation of a given query with issuer x_1 and query point x_2. Having obtained the query, the node responsible for x_2 answers with probability $1-p$. If it does not answer, the issuer generates negative feedback on the first forwarder and a feedback notification that is negative (see Subsection 5.4). The feedback notification is forwarded, just as queries are, with probability $1-p$ per step. In other words, our model reflects that feedback notifications may get lost, just as queries are.

Given the distance between issuer and answerer, we distinguish three cases: If the issuer itself is able to answer the query ($dist^{n,d}(x_1,x_2)=0$), no feedback is generated. If the query is answered by a direct neighbor ($dist^{n,d}(x_1,x_2)=1$), the query is transmitted with certainty, the answer occurs with $1-p$, and the respective feedback is generated with certainty again. If the distance is greater than 1, we have to take three points into account:

- A query issued by x_1 with query point x_2 is forwarded from x_1 to x_2 with probability $(1-p)^{dist^{n,d}(x_1,x_2)-1}$. This leads to the first factor in the third row of Equation 9.
- The node responsible for x_2 answers the query with probability $1-p$ (second factor).
- The resulting feedback notification arrives at the last node before x_2 in the chain of forwarders with probability $(1-p)^{dist^{n,d}(x_1,x_2)-2}$, once it is generated (third factor). That node generates feedback for having answered the query.

Putting everything together, Equation 9 is the probability of obtaining positive feedback items for having answered a query forwarded over δ hops, and Equation 10 is the average number of positive feedback items.

$$f_{10}^{aux}(p,\delta)=\begin{cases}0 & \text{if}\delta=0\\ 1\cdot(1-p) & \text{if}\delta=1\\ (1-p)^{\delta-1}\cdot(1-p)\cdot(1-p)^{\delta-2} & \text{if}\delta>1\end{cases} \tag{9}$$

$$h_{answer,pos}^{n,d}=\iint_{x_1,x_2\in[0;1]^d} f_{10}^{aux}(p,dist^{n,d}(x_1,x_2))dx_1dx_2 \tag{10}$$

Equation 12 is similar. It returns the average number of negative feedback items for not having answered queries. The node responsible for x_2 does not answer with probability p.

$$f_{12}^{aux}(p,\delta)=\begin{cases}0 & \text{if } \delta=0\\ 1\cdot p & \text{if } \delta=1\\ (1-p)^{\delta-1}\cdot p\cdot(1-p)^{\delta-2} & \text{if } \delta>1\end{cases} \tag{11}$$

$$h_{answer,neg}^{n,d}=\iint_{x_1,x_2\in[0;1]^d} f_{12}^{aux}(p,dist^{n,d}(x_1,x_2))dx_1dx_2 \tag{12}$$

The probability to obtain feedback items for having (not) answered a message decreases with the distance. Other nodes may drop the feedback notification instead of forwarding it. This is why our protocol incorporates not only feedback for answering queries, but for forwarding as well.

Let us now look at feedback for not forwarding. Peers cannot directly observe peers not forwarding queries. If a query does not return a result in time, each peer of the forwarding chain obtains a negative feedback notification message. Again, the formula has to consider three aspects: (1) forwarders may ignore the query message, (2) the peer responsible for x_2 may not answer it, and (3) the feedback notification can get lost. Negative feedback is generated on all peers before the answerer, i.e., for queries transmitted over distances less than 2, no such feedback is generated. Equation 13 reflects this. Then the average number of negative feedback per node per round is the sum of the probabilities over all nodes that might not have processed the query (Equation 14).

$$f_{14}^{aux}(p,\delta)=\begin{cases}0 & \text{if } \delta<2\\ (1-p)^{\delta-1}\cdot p\cdot\left(1+\sum_{j=1}^{\delta-2}(1-p)^{j-1}\right) & \text{if } \delta\geq 2\end{cases} \tag{13}$$

$$h_{forward,neg}^{n,d}=\iint_{x_1,x_2\in[0;1]^d}\sum_{i=1}^{dist^{n,d}(x_1,x_2)} f_{14}^{aux}(p,i)dx_1dx_2 \tag{14}$$

Formula 16 for positive feedback is similar but simpler. This is because we only need to look at the case where all nodes forward and answer the query.

$$f_{16}^{aux}(p,\delta)=\begin{cases}0 & \text{if } \delta<2\\ (1-p)^{\delta-1+1}\cdot\left(1+\sum_{j=1}^{\delta-2}(1-p)^{j-1}\right) & \text{if } \delta\geq 2\end{cases} \tag{15}$$

$$h_{forward,pos}^{n,d}=\iint_{x_1,x_2\in[0;1]^d} f_{16}^{aux}(p,dist^{n,d}(x_1,x_2))dx_1dx_2 \tag{16}$$

According to our protocol (cf. Figure 4-4), generating feedback occurs with a certain probability, allowing to differentiate between the different kinds of violations of the protocol. The numbers of feedback items actually generated are as follows:

$$\begin{aligned}\hat{h}_{forward,pos}^{n,d}&=h_{forward,pos}^{n,d}\cdot q_{forward}\\ \hat{h}_{forward,neg}^{n,d}&=h_{forward,neg}^{n,d}\cdot q_{forward}\cdot q_{pn}\\ \hat{h}_{answer,pos}^{n,d}&=h_{answer,pos}^{n,d}\cdot q_{answer}\\ \hat{h}_{answer,neg}^{n,d}&=h_{answer,neg}^{n,d}\cdot q_{answer}\cdot q_{pn}\end{aligned} \tag{17}$$

Given this, we have derived formulae for the expected numbers of feedback items for forwarding and answering. Feedback is also generated when a node has carried out a ProW. $\hat{h}_{ProW,pos}^{n,d}$ stands for the expected number of this kind of feedback items generated per round. We will derive a closed formula in what follows. When extending the protocol to cope with other kinds of adverse behavior, there will be more reasons for generating negative (or positive) feedback. The analysis steps that follow can take these kinds of feedback into account without difficulty.

How likely is a ProW request? As a first step, we estimate the expected lifetime of a feedback item in a feedback repository. The expected number of feedback items on a node generated per round is $\sum_x \hat{h}^{n,d}_{x,pos} + \sum_x \hat{h}^{n,d}_{x,neg}$, with $x \in \{forward, answer, ProW\}$. The larger the sum, and the smaller the size s of the repository, the smaller is the expected lifetime ttl of feedback items. The formula is

$$ttl = \frac{s}{\sum\limits_x \hat{h}^{n,d}_{x,neg} + \sum\limits_x \hat{h}^{n,d}_{x,pos}} \quad (18)$$

where the expected lifetime is given in numbers of rounds. The probability that the rating of a random feedback item in the feedback repository is negative or positive is as follows:

$$p^{n,d}_{neg} = \frac{\sum\limits_x \hat{h}^{n,d}_{x,neg}}{\sum\limits_x \hat{h}^{n,d}_{x,pos} + \sum\limits_x \hat{h}^{n,d}_{x,neg}} \quad (19)$$

$$p^{n,d}_{pos} = \frac{\sum\limits_x \hat{h}^{n,d}_{x,pos}}{\sum\limits_x \hat{h}^{n,d}_{x,pos} + \sum\limits_x \hat{h}^{n,d}_{x,neg}} \quad (20)$$

Given these probabilities, the following questions arise: What is the expected value of the reliability coefficient? How likely is it that the coefficient is below t? In other words, how likely is it that a cooperative node is deemed unreliable by its neighbors? Clearly, the value of the reliability coefficient follows the binomial distribution $B(s, p^{n,d}_{pos})$. Then the following holds:

$$E(\text{reliability coefficient}) = s \cdot p^{n,d}_{pos} \quad (21)$$

$$\begin{aligned} p^{n,d}_r &= P(\text{reliability coefficient} < t) \\ &= \sum_{i=0}^{t-1} \binom{s}{i} \cdot \left(p^{n,d}_{pos}\right)^i \cdot \left(1 - p^{n,d}_{pos}\right)^{s-i} \end{aligned} \quad (22)$$

Now we can determine the number of ProW requested from a cooperative peer. It depends on the probability of having a certain reliability coefficient, and the number of ProW needed to reach threshold t:

$$h^{n,d}_{ProW} = \sum_{i=0}^{t-1} \binom{s}{i} \cdot \left(p^{n,d}_{pos}\right)^i \cdot \left(1 - p^{n,d}_{pos}\right)^{s-i} \cdot \left\lfloor \frac{t-i}{q_{ProW}} \right\rfloor \quad (23)$$

Thus the number of positive feedback items created on average is $\hat{h}^{n,d}_{ProW,pos} = h^{n,d}_{ProW,pos} \cdot q_{ProW}$. Finally, the expected overall cost of a node per round is the frequency of the various tasks, multiplied with their respective costs.

$$\begin{aligned} E(\text{cost}) = h^{n,d}_{answer} \cdot c_{answer} &+ h^{n,d}_{forward} \cdot c_{forward} \\ &+ h^{n,d}_{ProW} \quad \cdot c_{ProW} \end{aligned} \quad (24)$$

Nodes that cooperate less. Now consider a node with a private failure rate q, i.e., defections and system failures. In what follows, we refer to a node with a failure rate q larger than p as *uncooperative*. How much feedback can it expect per round? The probability that a query is forwarded or answered changes from $1 - p$ to $1 - q$. Because all other nodes forward with probability $1 - p$ as before, the probability that a feedback notification reaches the predecessor of the node remains unchanged. Therefore, we adapt the existing formulae for having forwarded or answered by factor $(1 - q)/(1 - p)$. So the number of positive feedback items an uncooperative peer can expect is:

$$\begin{aligned} \tilde{h}^{n,d}_{forward,pos} &= h^{n,d}_{forward,pos} \cdot \tfrac{1-q}{1-p} \\ \tilde{h}^{n,d}_{answer,pos} &= h^{n,d}_{answer,pos} \cdot \tfrac{1-q}{1-p} \end{aligned} \quad (25)$$

The peer does not forward or answer queries with probability q, instead of p. So we adapt Formulae 12 and 14 by inserting factor q/p:

$$\begin{aligned} \tilde{h}^{n,d}_{forward,neg} &= h^{n,d}_{forward,neg} \cdot \tfrac{q}{p} \\ \tilde{h}^{n,d}_{answer,neg} &= h^{n,d}_{answer,neg} \cdot \tfrac{q}{p} \end{aligned} \quad (26)$$

The formulae for the amount of feedback actually generated are analogous to (17).

Costs of joining the CAN. Finally, we quantify the costs of joining the CAN. Obviously, a new node must carry out some ProW before being able to issue queries. When the node carries out a ProW, its neighbors will share information on this. Thus, a new node must take the following costs:

$$E(\text{cost}') = \frac{t}{q_{ProW}} \cdot c_{ProW} \quad (27)$$

7 Discussion

Given this cost model, we are interested in the impact of the various parameters and the interdependencies between them. In what follows, we limit the discussion to results that we find most interesting. We have used numerical methods to interpret the formulae.

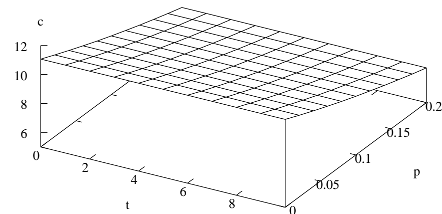

Fig. 8 Costs for forwarding and responding for cooperative nodes.

First we look at cooperative nodes. Their costs depend on the global failure probability p and the reliability threshold t. Figure 8 graphs the expected costs c for answering and forwarding queries. Figure 9 shows the expected ProW cost. Parameter p ranges from 0 to 0.2. Higher values for p are not meaningful, in contrast to, say, existing file-sharing systems. Since cooperative behavior dominates, as we will show, p will be small. The values of the other parameters are as in Table 2. Figure 8 shows that a high global failure probability reduces the costs of forwarding/answering queries. This is because queries will get lost on the way from the issuer

to the answerer. But this may go along with high ProW cost for all peers (Figure 9). Thus, ProW cost may account for a significant share of the cost of a cooperative node in a CAN with a high global failure probability, at least if the parameter settings have not been chosen carefully. We see that for $t = 0$ the ProW costs are independent of p. This makes sense: A node does not need to carry out a ProW, since it is deemed reliable in any case.

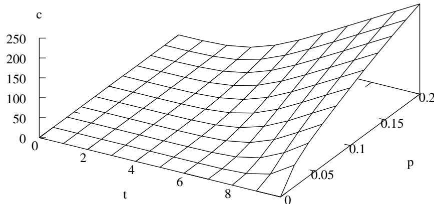

Fig. 9 Costs for answering ProW for cooperative nodes.

Let us now look at the costs of nodes that are uncooperative. Figure 10 plots the total cost c of an uncooperative node as a function of its failure rate q and threshold t. The failure rate of an uncooperative peer, which we have chosen arbitrarily for this particular plot, is $q = p \cdot 2$. In reality, an uncooperative peer is not likely to have such a low failure rate, only marginally higher than the average rate. But this graph is supposed to show that a small difference in the failure rates already affects the total costs of uncooperative peers significantly. Let us first look at the case where t is small. Every node qualifies as reliable, even though the feedback repositories might contain hardly any positive feedback item. In this case, costs are low since a node does not have to carry out any ProW. According to Figure 8, costs even slightly decrease with increasing q. The reason is that the node now processes fewer queries and therefore has lower cost. But according to Figure 10, this effect is insignificant, compared to ProW costs when t is high. In other words, if t is too low, it pays off to be uncooperative. On the other hand, Figure 9 tells us that large t penalize cooperative nodes.

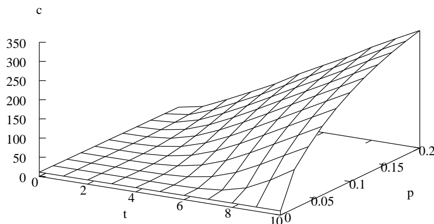

Fig. 10 Total costs for uncooperative nodes.

Fortunately, Figure 11 tells us that there are parameter settings where the discrimination c' between cooperative and uncooperative nodes is good and costs for cooperative peers remain acceptable. Figure 11 shows the ratio of the total cost of a peer with failure rate q and of one that is cooperative ($q = p \cdot 2$). The z-axis is the total cost of an uncooperative peer with failure rate q divided by the total cost of a cooperative peer. For the protocol to work, the ratio should always be significantly larger than 1. With the parameter settings chosen for the experiment, this does not always hold, e.g., for small t. Thus, to achieve the discrimination of uncooperative peers sought, it is necessary to choose the parameter settings more carefully. On the other hand, it seems that one can achieve the discrimination envisioned: There is a large part of the domain where the function value is larger than 1, and, according to Figure 9, the ProW costs of cooperative peers are moderate.

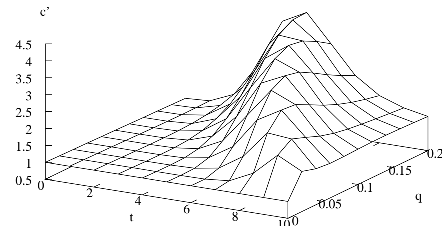

Fig. 11 Discrimination between cooperative and uncooperative nodes.

Having said this, the crucial questions are: What is an optimal setting of the endogenous parameters? How does the overall system behavior look like with such a setting? To answer these questions, we must clarify first what we mean with 'optimal'. Two objectives come to mind: (1) Discrimination against nodes with a higher failure rate should be as good as possible. (2) The additional overhead of cooperative nodes should be minimal. – Obviously, we cannot achieve both objectives at the same time: We can achieve a good discrimination against nodes that tend to be uncooperative by being strict, e.g., high value of threshold t, higher weight for negative feedback items, etc. But this means that cooperative nodes are more likely to have to carry out ProW, since they also obtain negative feedback. Their costs would increase. This would run counter to Objective (2). Hence, we wonder: Are there system states with good discrimination as well as moderate overhead of cooperative nodes?

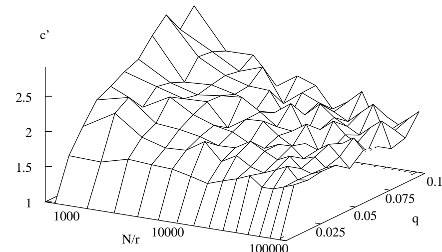

Fig. 12 Discrimination, with overhead bounded.

Figure 12 shows the maximal discrimination against uncooperative nodes for a given overhead. In more detail, we require that the overhead of cooperative nodes to work off ProW requests must be less than 10%, compared to the cost of forwarding and answering queries. '10%' is arbitrarily chosen; we want to show that even small additional expenses

from cooperative nodes lead to an acceptable discrimination. Other values yield similar effects.

Example 6 Suppose that the expected overall cost of a cooperative node is 70 for a certain setting of the endogenous parameters. The expected ProW cost is 20. The ProW cost as a ratio of the overall cost is 20/70. This is more than 0.1, which is our upper bound. Thus, we would ignore this setting and search for an other one.

Given this bound, Figure 12 shows the maximal discrimination that is possible, i.e., the overall cost of an uncooperative node divided by the overall cost of a cooperative node. Because the protocol depends on the path lengths, the x-axis is the number of zones in the CAN. This would be the number of peers if there was no replication. The y-axis is the failure rate q of the uncooperative node. Again, the private failure rate is set to $q = p \cdot 2$. Parameters to be optimized are t, $q_{forward}$, q_{answer}, q_{pn}, and q_{ProW}. From our cost model it follows that $c_{forward}/c_{answer} = q_{forward}/q_{answer}$. The values of $c_{forward}$, c_{answer}, and c_{ProW} are as in Table 2. $c_{forward}$, c_{answer} and c_{ProW} are exogenous parameters (cf. Subsection 5.1).

Figure 12 shows that the values are always larger than 1, i.e., the discrimination sought does occur! The discrimination is very good for small and medium values of N and small global failure probabilities p. However, it decreases with increasing N. Note that the protocol works for CAN with many nodes as well, as long as the path lengths are short enough. Choosing r and d appropriately guarantees that. – The curve is not as smooth as one might expect it to be. The reason is that the domains of some parameters are discrete. In particular, t must be an integer. The result from Figure 12 is positive since the discrimination envisioned does take place, with realistic settings of the various parameters. The threshold value for the additional cost of a cooperative node is very moderate (0.1). The values of q are conservative as well. If for instance $q = 0.1$, this means that the node still processes 90% of the incoming queries as requested. Discrimination against nodes with a higher failure rate is much more distinct.

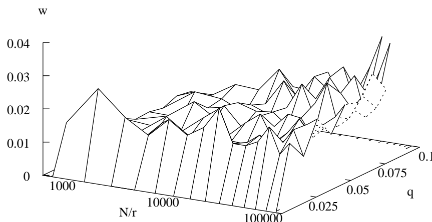

Fig. 13 Overhead, with discrimination bounded.

Figure 13 is based on the inverse perspective. Now the constraint is a minimum discrimination of 1.2 for partly uncooperative nodes. In other words, the expected total cost of a node with failure rate $q = p \cdot 2$ must be 20% more than the expected total cost of a cooperative node. Again, 1.2 is an arbitrary (bot moderate) value. Given this constraint, we look for settings where the expected ProW costs of cooperative nodes are minimal. Figure 13 shows the expected ProW cost w of a cooperative node divided by its expected total cost. In most cases, these additional costs are pleasingly low. They increase with increasing p and increasing N, but stay within a tolerable range. Thus, our protocol yields a good discrimination of partly uncooperative nodes with moderate effort.

In what follows, we investigate the influence of the remaining parameters. In particular, can they improve discrimination without increasing ProW costs? The role of Parameter s, the size of the feedback repository, seems to be evident: The larger s, the easier for a peer to collect evidence for cooperativeness and uncooperativeness, and the larger the expected discrimination. The opposite perspective is that small values of s might not result in good discrimination, at least when the bound on the overhead is small.

Figure 14 graphs the optimal discrimination for a given overhead of 10% and the private failure rate q (again, $p = q/2$), for different values of s. The grey color highlights values of s and q which do not allow to meet the overhead constraint. The figure reveals a positive result: there are always settings that fulfill the overhead bound, except for tiny repository sizes and very large failure rates.

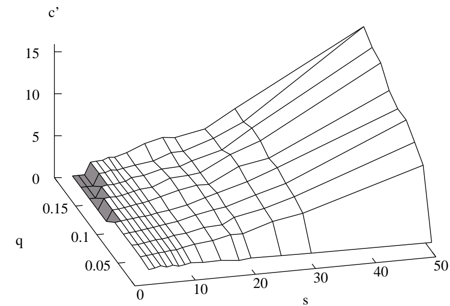

Fig. 14 Discrimination for different sizes of the repository.

Another parameter is c_{ProW}, the cost of a proof of work. It specifies the amount of work a node has to carry out in order to signal its willingness to cooperate. Our expectations regarding this parameter are as follows: if the ProW is too expensive, this will incur significant additional costs for cooperative nodes. If the ProW is too cheap, the number of ProW requests might be high, and this might bring down query performance.

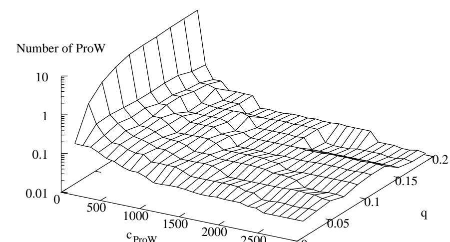

Fig. 15 Number of proofs for uncooperative peers.

Figure 15 plots the number of ProW requested from uncooperative nodes for different ProW costs and different failure rates, again with a ProW overhead bound of 10% for cooperative nodes. It meets our expectations in the following respect: low ProW costs incur a high rate of ProW requests and vice versa. But an unexpected result is that the number of ProW requested is exponentially decreasing. The inflection point is near a cost value of 200. This has an important implication: the problem of different private costs (cf. Section 5.1) goes away if ProW costs are high. Namely, very small intervals along the 'Number of ProW' axis relate to large intervals along the c_{ProW}-axis. Consequently, our protocol will remain effective in heterogeneous settings where peers run on different hosts, endowed with different capacities for ProW computation.

8 Experimental Evaluation

Having carried out an extensive model-based analysis, we want to verify that our reliability-aware CAN protocol is operational under real-world assumptions as well. We have evaluated our protocol by means of numerous experiments. They address the following issues:

- Does an implementation show the behavior predicted by our model, i.e., is our model correct?
- How restrictive are our assumptions behind the algebraic model (some of which are necessary to keep the formal analysis manageable)? Does the protocol remain operational if we drop assumptions like regular zones or completely even distribution of queries issued at steady rates?
- How sensitive is the system regarding small changes of the environment? In a living system, peers may alter their behavior or join and leave the CAN. Thus, characteristics like the failure probability, the number of peers, or the length of forwarding paths may change over time. In other words, assume that we have found the optimal values of the endogenous parameters for a given setting of the exogenous parameters. Now the exogenous parameters change. Do those values of the endogenous parameters still result in good discrimination and acceptable overhead?

We have performed all experiments on a Linux cluster of 32 loosely coupled workstations equipped with 2 GHz CPU, 2 GB RAM and 100 MBit Ethernet each. Here we ran a Java-based CAN implementation which we have implemented ourselves. We have extended it for logging and management purposes and stripped it from all unnecessary features (persistent storage, repair mechanisms, etc.) [7] provides a detailed description of our experimental setup. Given this setting, we ran a 4-dimensional CAN consisting of up to 160,000 nodes. The number of queries was 5,000,000. All parameters are set to optimal values (cf. Table 2) according to our model if not explicitly stated otherwise. To ensure that we are in steady state, we started measurements and statistics gathering only after an initialization period of 500,000 queries.

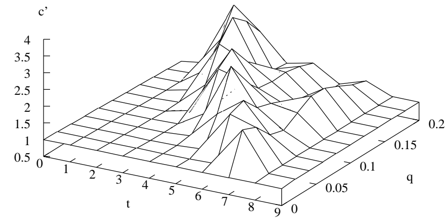

Fig. 16 Discrimination between cooperative and uncooperative nodes in the experiment.

Effectiveness. The most urgent issue is to show that our reliability-aware CAN protocol functions as envisioned – and as predicted by our model. Remember Figure 11, which graphs the calculated discrimination between cooperative and uncooperative nodes. It tells us that uncooperative nodes have to take significantly higher costs than cooperative ones if the threshold value is adjusted to the global and the local failure rate. We hope to see the same curve if we replay the analysis with our prototype (cf. settings in Section 7). Figure 16 shows the result of this experiment: leaving aside some smoothness issues, both curves are congruent!

Effects of model assumptions. In order to ease the algebraic analysis we introduced some assumptions which do not always hold in reality. We now want to show experimentally that these simplifications do not result in a model that is oversimplified, and predictions are realistic. In other words, the assumptions only hold for the derivation of the formal cost model, and we will show that they are not necessary to ensure functioning of the system. We have run experiments with 10,000 peers and a rate of 10% uncooperative nodes using a local failure rate of 20%. The first experiment reproduces the setting from the algebraic analysis, i.e., it relies on a completely regular zone-partition scheme and peers issuing queries at regular rates for equally distributed query keys. In subsequent experiments, we have replaced each of these simplifications with assumptions closer to the real world:

- The completely regular zone partitioning is replaced by a realistic zone scheme. It is the result of the split protocol of the original CAN (cf. Section 3). Peers are now responsible for zones of different sizes and have different numbers of neighbors.
- Instead of peers issuing the same number of queries at regular rates, we assign a factor to each peer whose distribution is Gaussian. It determines the number of queries issued in each round. The factor has mean value 1.0 and ranges from 0 to 2. Hence, the number of queries each peer issues per round remains 1.0 on average.
- The equally distributed query points are replaced by points whose distribution is Gaussian as well. Here, the mean value is the center of the 4d-hypercube of the key space. Thus, more queries go to the nodes in the center than the peers at the sides[5].

[5] Note that the key space is a d-torus. 'sides' is the part of the space close to the wrap-around from 1 to 0 and vice versa in each dimension.

We now turn off those simplifications, each one in isolation as well as all of them at the same time. Our expectations are minor deterioration in the discrimination and the cost measures, but the protocol should remain effective in each setting.

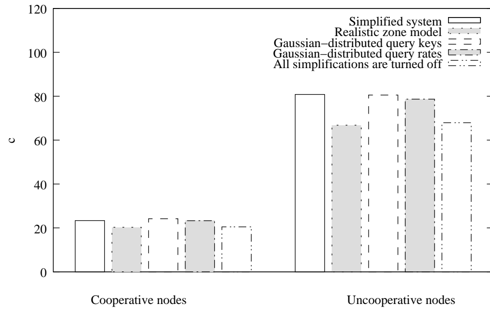

Fig. 17 Do our assumptions affect the costs?

Figure 17 plots the costs for cooperative and uncooperative nodes with some of these simplifications in place/turned off. It tells us that the largest change is due to the realistic assignment of zones to peers. This is reasonable: with the realistic partition scheme, zones are now different in size. With larger zones, a message forward covers a larger distance. If the number of uncooperative nodes remains constant, shorter paths decrease the global failure probability. Therefore, the ProW costs as well as the ones of forwarding messages decrease (cf. Figures 9 and 10). In contrast, sending queries at different rates to query points unevenly distributed in the key space does not make much of a difference. If a node issues queries at a rate different from the other nodes, say, at a higher rate, its feedback will be updated at a higher rate as well. But this does not affect the global behavior. The diagram confirms this.

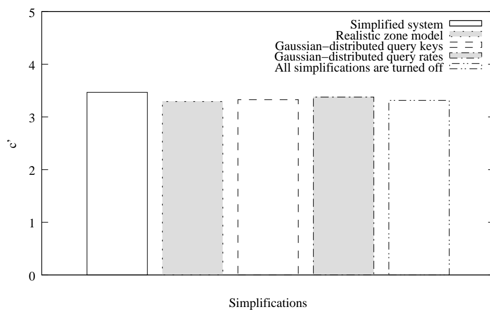

Fig. 18 Do our assumptions affect the discrimination?

Figure 18 shows the discrimination for the same settings. Removing the simplifications affects cooperative nodes and uncooperative ones as well. Thus, the discrimination remains nearly unchanged. Summing up, the result of these experiments is again positive. The protocol is effective, even when dropping the model assumptions.

Does the protocol penalize the neighbors of uncooperative nodes? A crucial point of the reliability-aware protocol is the following one: each peer in a chain of forwarders is penalized with negative feedback if the issuer does not receive an answer for its query. The rationale for this has been to guarantee that the node ignoring the query obtains negative feedback in any case. However, one might think that peers close to uncooperative nodes are 'punished' as well, and that they fare significantly worse than nodes which are not in the neighborhood of such peers. But the following experiment shows that our protocol copes well with these cases. We used a CAN consisting of 9,500 cooperative and 500 uncooperative ($q = 0.2$) nodes in a 4-dimensional key space.

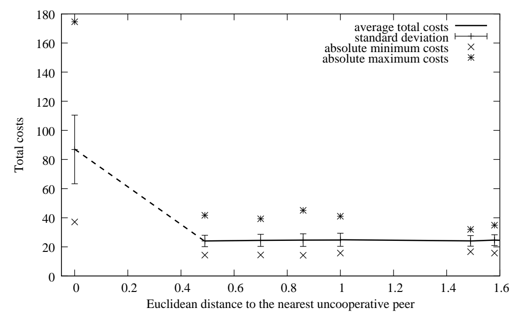

Fig. 19 Relation between costs and the distance to the closest uncooperative peer.

Figure 19 graphs the costs of nodes as a function of the distance to the closest uncooperative node. The costs displayed are the average costs of all nodes with the same distance to the closest uncooperative node, the standard deviation and the absolute maximum and minimum values. Because of the regular zone model there is a small finite number of distances to the closest uncooperative node. The distance metric is the Euclidean one, which is used for query forwarding as well. If the distance to the closest uncooperative node is 0, the node itself is uncooperative. If the distance is 0.5, the closest uncooperative node is an immediate neighbor. With a distance of 0.7, the peer and an uncooperative node have one corner in common, etc. The solid black line in Figure 19 connects the average cost values for each group of peers with the same distance.

The figure reveals the following information: on average, a node is not penalized at all for being close to an uncooperative node. A detail observation is that there is a significant spread in the costs, according to the standard deviations and the extrema values. This is due to the fact that Figure 19 considers only the closest uncooperative peer. In some cases, there may be several uncooperative peers quite close to the current node. In some cases, there may be only one uncooperative peer that is rather close. Another much more important finding from this experiment is that cooperative behavior dominates, except for extreme cases. However, the mean value minus the standard deviation of the costs of uncooperative peers are always larger than the maximum costs of cooperative peers, irrespective of their distance to the clos-

est uncooperative node. This is a positive result. Admittedly, one can construct special cases where this does not hold, e.g., a peer completely blocked in all directions by uncooperative nodes. But this is will virtually never happen under normal conditions. For example, the probability that all neighbors on one side of a peer are uncooperative is less than 0.5% for a 4-dimensional CAN with a rate of 50% uncooperative nodes, and this rate is unrealistically large. Further, replication brings down this probability to arbitrarily small numbers.

Rate of feedback creation. An important question is whether the number of feedback items in the repositories of the peers is sufficient. The amount of feedback generated depends on $q_{forward}$ and q_{answer}, which are external parameters coupled with the costs of forwarding and answering queries, and q_{ProW} and q_{pn}. The parameter q_{ProW} controls the number of feedback items generated for each proof of work. For instance, a value of $q_{ProW} = 1$ means that for each proof result the peer who has requested it will generate one positive feedback item. q_{pn} specifies the amount of feedback created if peers do not seem to follow the protocol. For example, $q_{pn} = 2$ states that for each query that remains unanswered each peer along the routing path receives twice the amount of feedback (of negative type), compared to the amount of positive feedback it would have obtained otherwise. Thus, the parameter controls how severely 'not forwarding' is actually punished.

Because of the complexity of the protocol, it is difficult to anticipate the effects of specific values of q_{ProW} and q_{pn}. We expect that a small number of feedback items generated for one ProW would lead to a high number of ProW requests. We further expect a large q_{pn} to penalize all nodes along the routing path with proofs for every message that has been rejected. Only experiments will shed light on these issues. Figure 20 graphs the number of proofs requested from a cooperative peer per round as a function of the two parameters. The rate of uncooperative peers is 50% ($q = 0.2$). We chose such an extreme rate for this particular plot in order to obtain results that expressive. Lower rates result in curves that are much flatter. Figure 20 shows that increasing q_{ProW} rapidly decreases the number of proofs requested (note that the scale of the z-axis is logarithmic). In contrast, small q_{ProW} and large q_{pn} lead to a very high number of proofs. As a result, the figure tells us that the range of parameter values that are viable is rather wide. Leaving aside extreme values below 0.5, the effect of both parameters is negligible, in particular when compared to threshold t. This result allows us to set the parameters to arbitrary, average values and to focus on the other parameters. Given the complexity of the system, this is helpful.

Scalability. Our model allows to adjust all parameters to optimal values if the degree of cooperativeness of the uncooperative nodes and the number of peers are known exactly. Both the algebraic analysis and the experiments we have presented so far indicate that our protocol is stable against changes in the behavior of the nodes. But what happens if the number of nodes changes? To investigate this, we calculated optimal parameter settings for a CAN of 10,000 nodes, and varied the network size from 81 to 160,000 nodes including a rate of 10% uncooperative nodes with a local failure rate of 20%.

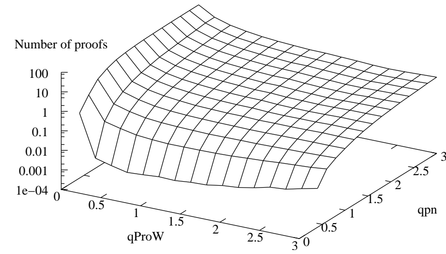

Fig. 20 Impacts of q_{pn} and q_{ProW}.

We anticipate that the best discrimination will occur at the number of nodes the parameters are optimized for, and increasing the number of nodes will increase the costs or decrease the discrimination. We will already declare success if the discrimination between cooperative and uncooperative nodes remains significant, and if the additional costs of cooperative nodes will not skyrocket for a wide range of CAN sizes.

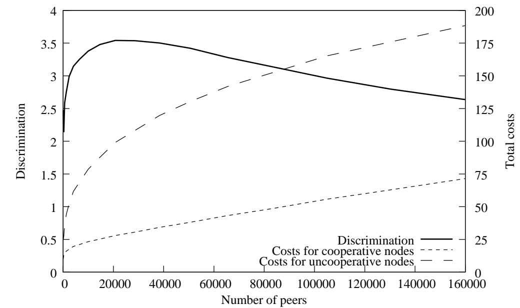

Fig. 21 Scalability of our protocol.

The graphs in Figure 21 show the discrimination between cooperative and uncooperative nodes (left axis and solid line) and the average costs for both kind of nodes (right axis and dashed line). The solid line is the quotient of the dashed lines. Our first prediction is true: the maximum of the discrimination is reached at approximately 10,000 nodes. The algebraic model tells us that increasing the number of nodes leads to increased path lengths for forwarding. Because the ratio of uncooperative nodes remains constant, i.e., the number of these nodes is increased as well, the effort for detecting uncooperative behavior will also increase. But our 'criterion for success' is met: the total costs for cooperative nodes remain significantly below the costs for uncooperative nodes for the numbers of CAN nodes observed. Furthermore, the graphs demonstrate that the discrimination continues to stay above a value of two, even for a number of nodes which is 16 times larger than the number of nodes used for the optimization. This is a result that we have not anticipated. Of course, adjusting the parameter values (e.g., by using self-tuning mechanisms or schemes from organic computing) would bring the discrimination to an optimum and would allow the protocol

to cope with an even larger number of nodes. But our experiment shows that this issue is not particularly urgent.

Figures 17 and 18 show that the model assumptions do not affect the global behavior at large. At most, they decrease the path length a little. But this shows that our model assumptions are rather conservative. We anticipate that the protocol remains scalable in the real world as well as in our experiments.

Why should a peer share feedback? With our protocol, a peer forwards feedback to its neighbors. But why should it do this? At first glance, this does not bring a direct advantage.

However, a simple 'Tit-for-Tat' extension of the protocol, i.e., a peer which does not disseminate feedback does not receive feedback in turn, changes things significantly, as we will show. A peer that does not exchange feedback with its neighbors will have to issue many more ProW requests than peers which do exchange feedback. The neighbors can easily detect this and refuse to carry out the ProW and exchange messages with that peer. Since a peer has at least $(3^d - 1)/2$ other neighbors suitable to forward a certain message to, this is feasible.

In the experiment that follows, we consider a simple Tit-for-Tat scenario, i.e., a peer which does not disseminate feedback does not receive feedback in turn. How does the rate of ProW requested differ between peers which share feedback and peers that are isolated?

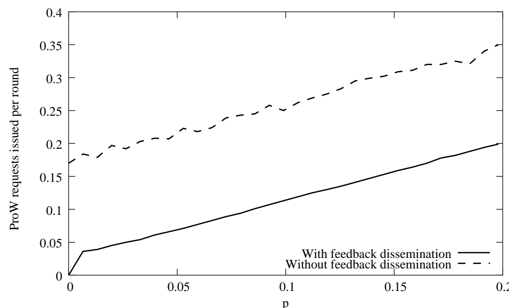

Fig. 22 ProW requested with and without disseminating feedback items between the peers.

Figure 22 graphs the outcome of an experiment where 10% of all nodes do not share feedback. In order to have expressive results, we varied the global failure probability (x-axis) from zero (no uncooperative peers, no ProW should be requested) to 20%. The y-axis is the number of ProW requested per round. Figure 22 tells us that peers which do not receive feedback from others (dashed line) issue many more ProW requests than peers sharing feedback (solid line). In settings with a low global failure probability, the difference of the rates is remarkably high. Therefore a node that wants to remain part of the network must decrease its extraordinary rate of ProW requests and share feedback.

9 Attacks

A crucial issue is the behavior of our protocol under different kinds of attacks, be they by free riders which want to save resources, be they by malicious nodes trying to harm other peers. In the presence of m-of-n data coding techniques and replication, we can neglect that other peers might be unable to access (key, value)-pairs in the zones of malicious nodes. Furthermore, we exclude problems arising from an incorrect implementation. In what follows, we look at different attacks and examine how they affect our system. Some of these attacks are well known from other contexts, others are specific to our protocol.

Spoof query results. In order to save the costs of data storage, an uncooperative node could send back a spoof query result.

On the other hand, the issuer of a query can verify the correctness of the result. This can take place in two ways. (1) In the case of replication, the issuer collects the query result from more than one node and forms a quorum. (2) In some applications, any peer can verify the correctness of a query result. For instance, if the CAN is used as a directory for object lookup or web-page annotations, a peer could always check if directory entries are valid. Our protocol as well as the formal analysis in Section 6 could be extended with negative feedback on spoof query results.

Man-in-the-middle attacks. A node could manipulate the content of messages it forwards to other nodes. [25] features a detailed discussion of this attack in CAN. The receiver of the message is supposed to think that the sender of the message gave false information. It might then give negative feedback.

We see three answers to this kind of attack: (1) The issuer could append a cryptographic signature to the message that allows the receiver to verify its correctness. Here, 'correctness' means that the message has not been tampered with since the send. This is in contrast to the notion of correctness used in the context of the previous attack. There, correctness referred to the quality of the query result from an application point of view. (2) The issuer could send multiple replicas along disjoint paths. This would require only a minor change of our routing protocol. (3) With our protocol, penalization with negative feedback extends to all peers in a chain of forwarders. As an extension of this idea, the forwarders of a query result that has been tampered with during forwarding will obtain negative feedback as well. Peers that modify query results repeatedly will then end up with a standing that is worse than the one of average cooperative peers.

Dissemination of spoof feedback. With our protocol, a peer accepts feedback only if

- its forwarder is deemed reliable, and
- it does not rate the forwarder itself.

In addition, a peer should accept feedback only if

- the frequency of feedback being generated and forwarded is comparable to the rate of other peers, and

– each feedback item contains plausible values, e.g., feedback comes from peers which could have made observations on the feedback subject in the first place.

Given this, what is the impact of a single peer disseminating spoof feedback?

Assume a peer tries to harm one of its neighbors by disseminating false negative feedback about it. The number of neighbors of a peer with regular partitioning of the key space is $a = 3^d - 1$. In the worst case, the malicious peer always generates negative feedback. How does the probability that a random feedback item whose subject is the attacked peer is positive change? It becomes $p_m = \frac{p_{pos} \cdot (a-1)+0 \cdot 1}{a}$.

Using the setting from Table 2 in a CAN with a global failure probability of 5%, Equation 23 tells us that the average number of ProW requested from a cooperative peer will rise from from 0.0098 to 0.0114 per round. We have conducted experiments which confirm this value. Here, the increase is even smaller than the standard deviation ($\sigma = 0.0068$). Summing up, the impact of a single peer which disseminates spoof feedback is small. Further, a peer can only attack its direct neighbors in the key space.

Collaboration attacks. Compared to the last attack, malicious peers that collude are more 'promising'. Think of a setting with the protocol extensions described in the context of the last attack and suppose that there is a group of colluding peers. Each of these peers only forwards requests from other colluding peers, and these peers steadily generate positive feedback about each other at a normal rate. As a prerequisite to do so, the colluding peers must have adjacent zones in the key space. Thus, the attackers must have tampered with the feedback mechanism as well as with the join protocol (cf. Section 3). The following questions arise: How many peers are necessary to form a group that allows to benefit from the CAN without participating in the work (or with less work than regular)? How does the CAN react to such a group of colluding peers?

We hypothesize that a large number of peers is able to keep the number of positive feedback items above the threshold t in the repositories of other peers. In addition, we suppose that the number of ProW requested from neighbors of the colluding peers will rise to the level of an uncooperative peer. We now want to validate these expectations by experiments. We again use the parameter values from Table 2, i.e., $N = 10,000$, each peer has $a = 80$ contacts, and the threshold for reliability is $t = 4$. We run experiments with different numbers of colluding peers which form one tight group in the key space. In order to make it more difficult to distinguish between reliable peers and attackers, 10% of the peers are uncooperative with a local failure probability of 50%.

Figure 23 shows the results of the experiments. The x-axis is the size of the group of colluding peers, the y-axis shows the average total costs of cooperative, uncooperative and colluding peers as well as the ones of the cooperative neighbors of the colluding peers. The result is surprisingly positive: in the range examined, the average costs of an attacker are always higher than the costs of a cooperative node, and the effect on cooperative nodes receiving spoof feedback from colluding peers are only moderate! A further analysis (not explicitly described here) has shown that the attackers at the 'outside' of the group, i.e., colluding peers with many cooperative neighbors receive much more negative feedback than spoof positive feedback disseminated by other colluding peers. Obviously, colluding peers completely surrounded by other peers from the coalition do not have to perform any ProW. But in our four-dimensional setting each peer has 80 neighbors. For instance, assume that we want to construct a coalition where 10 peers are completely surrounded by other peers from the coalition. Then the coalition must contain at least 270 peers. Thus, to profit from that effect significantly, the groups of peers that collude would have to be huge.

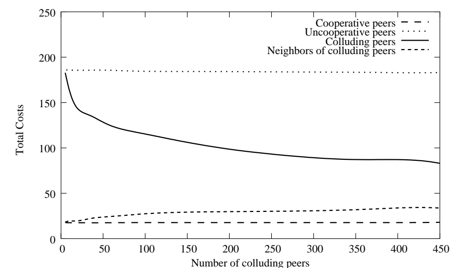

Fig. 23 Impacts of collaboration attacks.

Proxy attack. Another way to decrease the cost of participation is to have one peer as a proxy for many nodes. The proxy participates cooperatively in the work. But it issues queries as a substitute for many nodes which are not registered in the CAN. It returns the query results to the original issuers.

From our perspective, it is subject to a philosophical discussion whether this is legitimate behavior or an attack. Clearly, a lot of selfish users can benefit from the CAN at the cost of operating a single node. On the other hand, the peer that is visible in the CAN behaves cooperatively, and the integrity of the CAN is not affected. Furthermore, think of nodes located in a private network. A proxy on the firewall could be the only way out. A potential solution to the problem seem to be load balancing mechanisms for P2P data structures. The design of such mechanisms in a tamper-proof way is a major issue that goes well beyond the scope of this current article.

Infrequent queries, batching of queries. From the perspective of a single peer, it might be more efficient not to cooperate, but issue a batch of queries from time to time and carry out one or several ProW in order to obtain most query results. In other words, let us know drop the assumption of frequent queries that cannot be issued as a batch (cf. Subsection 5.1). How does the dominant strategy now look like?

The costs of an entirely untrusted node to become trusted by carrying out proofs of work are $c_u = c_{ProW} \cdot t / q_{ProW}$. In one round, a cooperative peer must bear the following costs: $c_c = c_{forward} \cdot h^{n,d}_{forward} + c_{answer} \cdot h^{n,d}_{answer} + c_{ProW} \cdot h^{n,d}_{ProW}$.

Therefore, the break-even between cooperative nodes and nodes that issue queries as a batch is at c_u/c_c rounds. Using the values from Table 2, the break-even lies at 24 rounds, i.e., a node that can wait for more than 24 rounds can be better off by batching queries and complying with ProW requests instead of behaving cooperatively all the time. Clearly, it is application-specific if a node can wait that long to have its queries answered. There are applications where such long waits are not practical (cf. Subsection 5.1).

To sum up this discussion on potential attacks, we notice that: (1) Leaving aside extreme scenarios, e.g., attackers have virtually unlimited resources, our protocol remains operational in the presence of malicious nodes. (2) Individual uncooperative peers or medium-sized groups of peers are unlikely to profit from attacks on the feedback mechanism. An exception is the proxy situation, which does not necessarily count as an attack. (3) The presence of malicious nodes in the neighborhood does not significantly affect the costs of cooperative nodes. (4) Repair mechanisms and data coding techniques that already exist in the literature and that are applicable here as well avoid that data may be lost.

10 Conclusions

Free riding in P2P data structures is an important problem that has not received much attention so far. The problem is difficult because the degree of free riding of individual nodes is not readily observable, among other reasons. Further, there exists no centralized instance that might act as an authoritative coordinator. Payment mechanisms do not solve the problem, since they require such a centralized instance. In addition, infrastructure costs would simply be too high. Our solution is a new CAN protocol where peers generate and disseminate feedback, and only peers with mostly positive feedback have their queries processed right away. All other peers must provide proofs of work from time to time, depending on their reliability. This mechanism is necessary to deter defection. The protocol does not guarantee that cooperative nodes will not have to carry out any proofs of work. But this article provides a formal analysis and extensive experiments that show the following: the protocol can differentiate fairly well between cooperative and uncooperative peers, and additional costs of cooperative peers are moderate. This holds true for many realistic settings of the exogenous parameters. In other words, cooperative behavior dominates, at least under the assumptions listed in the body of this article.

Many issues remain for future work, including the design of protocols for other P2P data structures. Since the contact list of a peer is not restricted to its neighbors any more, feedback dissemination is a more difficult problem. The interdependencies with applications on top of the P2P data structures also require more attention. Finally, while our protocol is robust against most kinds of malicious behavior, there are variants that remain to be explored.

References

1. Aberer, K.: P-Grid: A Self-Organizing Access Structure for P2P Information Systems. In: CooplS '01: Proceedings of the 9th International Conference on Cooperative Information Systems, pp. 179–194. Springer-Verlag, London, UK (2001)
2. Aberer, K., Despotovic, Z.: Managing Trust in a Peer-2-Peer Information System. In: Proceedings of the Conference on Information and Knowledge Management (2001)
3. Adar, E., Huberman, B.: Free Riding on Gnutella. First Monday **5**(10) (2000)
4. Back, A.: Hashcash - A Denial of Service Counter-Measure. http://www.cypherspace.org/∼adam/hashcash/ (2002)
5. Buchegger, S., Boudec, J.Y.L.: Coping with False Accusations in Misbehavior Reputation Systems for Mobile Ad-hoc Networks. Technical Report IC/2003/31, EPFL (2003)
6. Buchmann, E., Böhm, K.: FairNet - How to Counter Free Riding in Peer-to-Peer Data Structures. In: Proc. of the International Conference on Cooperative Information Systems 2004, Agia Napa, Cyprus (2004)
7. Buchmann, E., Böhm, K.: How to Run Experiments with Large Peer-to-Peer Data Structures. In: Proc. of the 18th Int. Parallel and Distributed Processing Symposium, Santa Fe, USA (2004)
8. Buragohain, C., Agrawal, D., Suri, S.: A Game-Theoretic Framework for Incentives in P2P Systems. In: Proceedings of P2P2003, Linkoping, Sweden (2003)
9. Chang, B.E., et al.: Trustless Grid Computing in ConCert. Proceedings of Grid02, Berlin (2002)
10. Datta, A., Hauswirth, M., Aberer, K.: Beyond "Web of Trust": Enabling P2P E-Commerce. Proceedings of the IEEE Conference on E-Commerce, USA (2003)
11. Feldman, M., Lai, K., Stoica, I., Chuang, J.: Robust Incentive Techniques for Peer-to-Peer Networks. ACM Electronic Commerce, 2004. (2004)
12. Friedman, E., Resnick, P.: The Social Cost of Cheap Pseudonyms. Journal of Economics and Management Strategy **10**(2) (1998)
13. Garcia-Molina, H., Schlosser, M.T., Kamvar, S.D.: The EigenTrust Algorithm for Reputation Management in P2P Networks. Proceedings of the Twelfth International World Wide Web Conference (2003)
14. Golle, P., et al.: Incentives for Sharing in Peer-to-Peer Networks. Lecture Notes in Computer Science **2232** (2001)
15. Gribble, S.D., et al.: The Ninja architecture for robust Internet-scale systems and services. Computer Networks (Amsterdam, Netherlands) **35**(4) (1999)
16. Gummadi, K., et al.: The Impact of DHT Routing Geometry on Resilience and Proximity. In: Proceedings of the SIGCOMM 2003. ACM Press (2003)
17. Jakobsson, M., Juels, A.: Proofs of Work and Bread Pudding Protocols. In: Proceedings of the CMS'99, Leuven, Belgium (1999)
18. Kroell, B., Widmayer, P.: Distributing a Search Tree Among a Growing Number of Processors. Technical Report 1994PA-drt, Swiss Federal Institute of Technology, Zürich (1994)
19. Litwin, W., Neimat, M.A., Schneider, D.A.: LH* - Linear Hashing for Distributed Files. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, Washington, D.C. ACM Press (1993)
20. Marsh, S.: Formalising Trust as a Computational Concept. Ph.D. Thesis., Department of Mathematics and Computer Science, University of Stirling (1994)

21. Marti, S., et al.: Mitigating Routing Misbehavior in Mobile Ad Hoc Networks. In: Proceedings of the Conference on Mobile Computing and Networking. ACM Press (2000)
22. Padovan, B., et al.: A Prototype for an Agent based Secure Electronic Marketplace Including Reputation Tracking Mechanisms. In: Proceedings of the Hawaii International Conference on System Sciences (2001)
23. Ramaswamy, L., Liu, L.: Free Riding: A New Challenge to Peer-to-Peer File Sharing Systems. Proceedings of the 36th HICSS Conference, Hawaii (2003)
24. Ratnasamy, S., et al.: A Scalable Content-Addressable Network. In: Proceedings of the ACM SIGCOMM Conference. ACM Press, New York (2001)
25. Reidemeister, T., et al.: Malicious Behaviour in Content-Addressable Peer-to-Peer Networks. Proceedings of the Third Conference on Communication Networks and Services Research, Canada (2005)
26. Resnick, P., et al.: Reputation Systems. Commun. ACM **43**(12) (2000)
27. Rowstron, A., Druschel, P.: Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In: IFIP/ACM International Conference on Distributed Systems Platforms (2001)
28. Stoica, I., et al.: Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In: Proceedings of the ACM SIGCOMM Conference (2001)
29. Xiong, L., Liu, L.: PeerTrust: Supporting Reputation-Based Trust for Peer-to-Peer Electronic Communities. IEEE Trans. Knowl. Data Eng. **16**(7) (2004)
30. Yang, B., Garcia-Molina, H.: PPay: Micropayments for Peer-to-Peer Systems. In: V. Atluri, P. Liu (eds.) Proceedings of the 10th ACM Conference on Computer and Communication Security (CCS-03). ACM Press, New York (2003)