FairNet – How to Counter Free Riding in Peer-to-Peer Data Structures

Erik Buchmann, Klemens Böhm

Otto-von-Guericke Universität, Magdeburg, Germany
{buchmann|kboehm}@iti.cs.uni-magdeburg.de

Abstract. Content-Addressable Networks (CAN) manage huge sets of (key, value)-pairs and cope with very high workloads. They follow the peer-to-peer paradigm: They consist of nodes that are autonomous. This means that peers may be uncooperative, i.e., not carrying out their share of the work while trying to benefit from the network. This article deals with this kind of adverse behavior in CAN, e.g., not answering queries and not forwarding messages. It is challenging to design a forwarding protocol for large CAN of more than 100,000 nodes that bypasses and excludes uncooperative nodes. We have designed such a protocol, with the following characteristics: It establishes logical networks of peers within the CAN. Nodes give positive feedback on peers that have performed useful work. Feedback is distributed in a swarm-like fashion and expires after a certain period of time. In extreme situations, the CAN asks nodes to perform a proof of work. Results of experiments with 100,000 peers are positive: In particular, cooperative peers fare significantly better than uncooperative ones.

1 Introduction

Content-Addressable Networks (CAN [1]) manage huge sets of (key, value)-pairs and cope with very high workloads. A CAN is an example of the peer-to-peer (P2P) paradigm: it consists of *nodes*, a.k.a. *peers*. Peers are autonomous programs connected to the Internet. Nodes participate in the work, i.e., data storage and message processing in the context of CAN. At the same time they also make use of the system. In CAN this means that they may issue queries. As with any P2P system, the owners of the nodes bear the infrastructure costs.

Autonomy of the peers implies that there is no coordinator that checks the identity or intentions of nodes. Doing without a coordinator has important advantages, such as scalability and no single point of failure. But it also implies that peers may try to reduce the costs of participation. With conventional CAN protocols, nodes actually do take part in the work voluntarily, and a node can reduce its infrastructure dues by not carrying out its share of the work. In economic terms, the dominant behavior is free riding [2], and the situation is an instance of the Prisoner's Dilemma [3]. In the context of CAN, free riding means ignoring incoming messages that relate to queries issued by other nodes. This can be achieved by tampering the program, blocking the communication, etc. In our terminology, such nodes are *unreliable* or *uncooperative*. It is very important to rule out this kind of behavior. The motivation of the owners of cooperative nodes will decline rapidly otherwise. From a technical perspective, the CAN might fall apart if some peers do not cooperate, and it might not be able to evaluate many queries.

Existing work does not solve these problems. Related work in mobile ad-hoc networks [4–7] assumes that adjacent nodes can eavesdrop traffic – detecting uncooperative behavior is easy. Others have proposed micropayments, public-key infrastructures, and certified code in similar contexts [8, 9]. But infrastructure costs would be unreasonably high, and the resulting system would not be P2P any more. Related work on trust management in P2P systems does not scale to many nodes ($>$ 100,000), or does not deal with message forwarding [10, 11].

This article proposes a CAN protocol that renders free riding unattractive, and focuses on the evaluation of queries. The protocol envisioned has the following objectives: (1) Nodes deliver the results of queries of another node only as long as it is cooperative. (2) At the same time, they should not rely on a node that has not proven its cooperativeness. (3) All this should not affect cooperative nodes, except for some moderate overhead.

Designing such a protocol is not obvious: There is no central ’trusted authority’, so nodes must rely solely on past interactions of their own or of reliable nodes they know. Further, CAN are supposed to work with large numbers of nodes, e.g., 100,000 or more. Finally, the behavior of peers may change over time. Our solution is a CAN protocol that establishes logical networks of peers within the CAN. Nodes give positive feedback on nodes that have performed useful work. Feedback is distributed piggybacked on ’regular’ messages in a swarm-like fashion and expires after a certain period of time. Query results are sent via chains of reliable nodes only. The effect is that the logical networks do not answer queries from uncooperative nodes.

Our evaluation is experimental and is directed towards one main question: Are the mechanisms proposed here effective, i.e., do they rule out uncooperative behavior, with moderate overhead? Our experiments show that the mechanisms do serve the intended purpose, and we have obtained these results for large CAN. In a network of 100,000 nodes, cooperative peers fare significantly better than partly or fully uncooperative ones.

Enforcing cooperation in distributed systems whose components are autonomous is a broad and difficult issue, and we readily admit that this article is only a first stab at the problem. Aspects not explicitly addressed include spoof feedback, spoof query results, and malicious behavior and application-specific issues. However, we think that our lightweight approach for reputation management is extensible, e.g., with negative feedback. This would allow to counter those kinds of adverse behavior effectively.

The remainder of this article has the following structure: After reviewing CAN in Section 2, Section 3 provides a discussion of cooperativeness in CAN. Section 4 introduces our reliability-aware forwarding protocol. Section 5 features an experimental evaluation. Related work is discussed in Section 6, and Section 7 concludes.

2 Content-Addressable Networks

Content-Addressable Networks (CAN [1]) are a variant of *Distributed Hash Tables (DHT)*. Alternatives to CAN differ primarily in the topology of the key space [12–14].

Each CAN node is responsible for a part of the key space, its *zone*. I.e., the node stores all (key, value)-pairs whose keys fall into its zone. This space is a torus of Cartesian coordinates in multiple dimensions, and is independent from the underlying physical network topology. In other words, a CAN is a virtual overlay network on top of a large physical network. In addition to its (key, value)-pairs, a CAN node also knows its neighbors, i.e., nodes responsible for adjacent parts of the key space.

A query in CAN is simply a key in the key space, its result is the corresponding value. I.e., a query is a message addressed by the query key. A node answers a query if the key is in its zone. Otherwise, it forwards the query to another node, the *target peer*. To do so, the peer uses *Greedy Forwarding*. I.e., given a query that it cannot answer, a peer chooses the target from its neighbors according to the following criteria: (1) The (Euclidean) distance of the key to the target in question is minimal. (2) The target node is closer to the key than the current peer. In what follows, we refer to the protocol described so far as *classic CAN*.

CAN, as well as any other DHT, are useful as dictionaries. In a file-sharing scenario, the CAN would store the locations of the files, and the files remain at their original locations. Other applications for DHT are annotation services which allow users to rate and comment web pages, or push services for event notification.

2.1 Enhancements to the Classic CAN

Greedy Forwarding in classic CAN sends messages from neighbor to neighbor. This causes a problem, at least when the key space is low-dimensional: The number of peers forwarding a certain message (*message hops*) is unnecessarily large. We have proposed in [15] that a peer does not only know its neighbors, but some remote peers as well. The so-called *contact cache* of the peer contains these remote peers. The contact cache is limited in size. In contrast to the neighbors, contacts may be out of date, and a peer may replace a contact by another one at any time. Furthermore, messages have an attachment. It contains contact information of the peers that have forwarded the message and of the peer that has answered it. A peer that receives such a message uses this information to update its contact cache. No additional messages are necessary to maintain the contact cache. Greedy Forwarding does not only take the neighbors into account, but the ones in the contact cache as well. Finally, Greedy Forwarding also works if information in the contact cache is outdated. In what follows, we refer to this variant of CAN as *enhanced CAN*.

[15] shows that even a small number of additional contacts decreases the average number of message hops significantly. For instance, a contact cache of size 20 (i.e., 20 peers in addition to the neighbors) reduces the number of hops in a CAN of 100,000 peers by more than $^2/_3$, assuming a real-world distribution of the keys of the queries and of the (key, value)-pairs.

Example 1: Peer S in Figure 1 is responsible for zone (0.3, 0.3 : 0.5, 0.5). Assume that it has issued a query with the key $(0.9, 0.9)$, i.e., it wants to retrieve the corresponding value. Since S is not responsible for this key, it uses Greedy Forwarding and sends the message to P_1. P_1 in turn is not responsible either and forwards the message to P_2, etc. Once the right peer R is reached, it returns the query result to S. In the enhanced CAN, the result is directly returned to the issuer of the query. $\square$

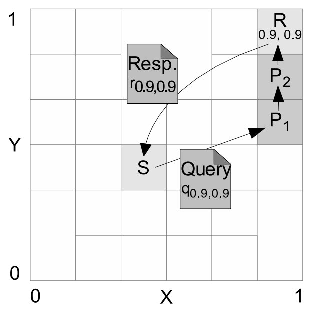

Fig. 1. Forwarding in Enhanced CAN.

3 Cooperation in CAN

In our terminology, a node that handles all incoming messages as expected is *cooperative*. A cooperative node answers the query if the key falls into its zone, or forwards it to another node that seems to be appropriate. From the point of view of another peer, the node is *reliable*. *Uncooperative* nodes in turn try to benefit from the network in a selfish way. In our context, uncooperative behavior is ignoring incoming messages that have to do with queries issued by other nodes.

Since uncooperative nodes hide their intentions and do not come up with statements like ”Connection Refused” or ”Host Unreachable”, repair mechanisms like Expanding Ring Search or Flooding [1] will not work. Furthermore, such nodes may spread falsified information to improve their standing. This implies that classic CAN might fall apart in the presence of uncooperative nodes.

Example 2: In a classic CAN with a percentage u of uncooperative peers that are not explicitly known, the probability p of forwarding a message via n peers is $p = (1-u)^n$. For example, in a network with 5% uncooperative peers, the probability to send a message via 10 nodes is less than 60%. The average path length in a network with a d-dimensional key space and c nodes is $l = (d/4)(c^{1/d})$ (see [1]). Given a key space with $d = 4$, $l = 10$ for 10,000 peers.

Now think of a CAN protocol that bypasses uncooperative peers when forwarding a query. Then the only peer that it cannot bypass is the peer responsible for the key of the query, so $p = 1 - u = 95\%$. Replication may improve the situation further, but this is beyond this article. □

A peer can estimate the reliability of a certain other peer if it has observed its behavior a couple of times. But frequently this is not the case. For instance, think of a new peer that has issued a query before it had a chance to prove its reliability. Therefore our protocol incorporates a *proof of work (ProW) protocol*: a node proves to another node that it has invested a certain amount of resources by providing some ”certificate” of resources spent [16, 17]. ProW can be seen as *entry fees* for the CAN, paid to one peer. A ProW is a mathematical problem that is hard to solve, but the solution is easy to verify. The ProW has nothing to do with the operations performed by the CAN. We for our part are not concerned with the design of ProW; we just deploy the concept. –

The rationale behind ProW is determent: With our protocol, an uncooperative peer will be more likely to carry out an expensive ProW. Hence, it is more economic to behave cooperatively.

The design of a reliability-aware CAN protocol depends on the attributes of the nodes and the characteristics of the applications. We make the following assumptions. These assumptions are quite similar to the ones behind other P2P protocols.

Application profile with frequent queries, small results. This article focuses on an application profile for P2P data sharing 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. It is acceptable if some (very few) queries remain unanswered. – Example applications are object lookup systems, annotation services, push services etc. These assumptions imply that sophisticated and expensive countermeasures [10, 9] against free riders are not applicable in our settings. We strive for lightweight mechanisms that must cope with a high rate of parallel queries and that make cooperation the *dominant* behavior.

Timely query results. Query results are needed in time, so it is infeasible to batch queries and issue them at once. – If we allowed peers to issue batches of queries, they could get by by behaving cooperatively from time to time only. Note that the sample applications mentioned in the previous paragraph fulfill this assumption as well.

Equal private costs. A general problem is that the cost of a node, regarding memory, network or CPU consumption, is private information. E.g., a peer connected with a dial-up modem is more interested in saving network bandwidth than one using a leased line. But observing the capabilities of other nodes is difficult. We leave this aside for the time being and assume equal private costs for all nodes. Our protocol could be extended to address different costs by using ProW of different extents.

Messages are not modified during forwarding. We assume that only the issuer of information can have falsified it. For example, a peer may create falsified feedback. But it is unable to intercept a response message and claim to be the peer who has provided the query results. In the presence of cryptographic signatures and the unlimited connectivity of the Internet, this is a realistic assumption: Each peer can ask the issuer of a message to verify its integrity. – This assumption allows us to come up with a protocol that rejects feedback from unknown or uncooperative sources.

No uncooperative behavior at application level. This article leaves aside misbehavior from the application perspective. For example, a node may want to prevent other nodes from obtaining access to a certain (key, value)-pair containing unfavorable information. It might try to accomplish this by running a DoS attack on nodes responsible for the pair. When looking at the storage level in isolation, such an attack consumes resources without providing any benefit. 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. In other words, our protocol does not deal with nodes that spend their resources for attacking the network, or that try to discredit a single peer, but behave reliably otherwise.

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 in two ways. (1) In the case of replication, the node collects the query result from more than one node and forms a quorum. (2) In some applications, any peer can verify the correctness of the 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. We do not expect any major difficulties when extending our protocol in this direction.

4 A Reliability-Aware CAN Protocol

In the context of our protocol, each peer decides individually if it deems another peer reliable, based on a set of observations from the past. We refer to such observations as *feedback*. Each node can only make observations on operations it is involved in. In our context, a peer accepts another peer as reliable or not – there are no shades in between. We settled for a simple reliability model because a sophisticated one (cf. [18]) would lead to a binary decision just as well. In addition, information about other nodes are always imperfect, hence a rich model would only mock a degree of accuracy that is not achievable in reality.

Our protocol has four aspects:

– Peers observe nodes and generate feedback (Subsection 4.2),
– share feedback with others (Subsection 4.3),
– administer feedback in their repository (Subsection 4.4),
– use feedback to bypass unreliable peers (Subsection 4.5).

4.1 Data Structures

Our implementation contains two classes *FeedbackRepository* and *Feedback*. They refer to classes already used in enhanced CAN, notably *ContactCache* and *Message*. Feedback objects bear feedback information. The node a Feedback object refers to is the *feedback subject*. Further, a Feedback object contains a *timestamp* and the ID of the peer that has generated the feedback, the *feedback originator*. Each node has one private FeedbackRepository object that implements its feedback repository. It stores t Feedback objects, for s_r peers each. It has methods for checking the reliability of a peer and for selecting Feedback objects to be shipped to other peers. Table 1 lists all relevant parameters. We discuss their default values in Subsection 5.1.

4.2 Generating Feedback

A peer assumes that another peer is reliable if its feedback repository contains at least t Feedback objects referring to that peer. Because peers may change their behavior, Feedback objects expire after a period of time (e). Thus, only a continuing stream of

Protocol-related Parameters

Symbol	Description	Default Value
t	reliability threshold; unit: number of Feedback objects	3
q	number of feedback objects generated for one forward	0.1
e	Feedback object lifetime; unit: experiment clock time units	$100,000$
b	maximum number of Feedback objects per message	20
s_r	size of feedback repository; unit: number of peers	100
s_c	size of contact cache; unit: number of peers	20

Experiment-related Parameters

Symbol	Description	Default Value
n	number of peers	$100,000$
d	dimensionality of the key space	2
u	percentage of uncooperative peers	5%

Table 1. Relevant Parameters.

positive feedback lets a peer be reliable in the eye of others. Feedback is generated in the following situations:

F1 *After joining the CAN, the new peer generates t Feedback objects for the peer who handed over the zone.*
F2 *After receiving an answer to a query, a peer generates one Feedback object for the node that has answered.*
F3 *After observing a message forward, the current peer generates one Feedback object with probability q.*
F4 *After having obtained a ProW, the receiver creates t Feedback objects for the peer that has delivered it.*

F2 acknowledges answering queries. Because forwarding messages consumes less resources than answering queries, F3 generates feedback only with probability q. Finally, providing a ProW (F4) or helping a new peer join the CAN (F1) are strong indications that the peer is cooperative. Because at least t Feedback objects are needed to deem a peer reliable, this is the number of Feedback objects created. In each case, the timestamp of the Feedback object is the current time, and the feedback originator is the current peer. The Feedback object is stored in the feedback repository of the current peer.

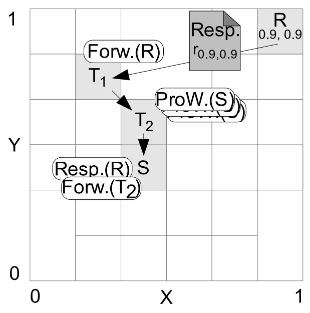

Fig. 2. Sources of feedback.

Example 3: Consider the situation depicted in Figure 2. Peer R answers a query issued by S. Because T_1 is the peer closest to S that R deems reliable, R sends it the message. According to F3, T_1 decides to create a Feedback object[1] with subject R, in order to acknowledge that R has forwarded the message. T_1 then forwards the message to T_2. In our example, T_2 does not generate feedback with subject T_1 because this only happens with probability q. The only next peer possible is S, but T_2 does not know if it is reliable or not. Therefore, T_2 asks for a proof of work. S returns this proof, so T_2 creates t Feedback objects with subject S and forwards the message to it. Finally S obtains its answer. It creates one Feedback object with subject R for answering, and one with subject T_2 for forwarding. $\square$

4.3 Disseminating Feedback Information

Sharing feedback between peers results in logical networks of peers that are *transitive*: one peer sees that another one performs useful work and spreads this information to others. To bound the overhead of our protocol, a node appends a small set of b Feedback objects to messages that it sends out anyhow.

Method *generateFeedbackAttachment* (Figure 3) determines an adequate set of Feedback objects to be attached. It is invoked with the outgoing message and the peer the message will be forwarded to. objectives of our feedback dissemination algorithm. Subsection 2.1 has pointed out that each peer needs a well-balanced set of *reliable* contacts to forward messages to. Each peer must be provided with a good set of feedback objects on peers in its contact cache. There are two locations where feedback is helpful:

(1) peers far away from the feedback subject who have the feedback subject in their contact caches, and
(2) neighbors of the feedback subject.

According to (1), method generateFeedbackAttachment first selects Feedback objects whose subjects have forwarded the current message. It starts with the node that has forwarded the message directly to it, and selects all feedback from its repository regarding that node. The procedure recurs with the next peer in the chain of the last

[1] Rounded rectangles stand for Feedback objects, located next to their originator, with the feedback subject in parentheses.

```
generateFeedbackAttachment(Message m, TargetPeer $P_t$) {
    FeedbackAttachment F := $\emptyset$;
    // (1) get Feedback objects about the last forwarders
    **forall** (p $\in$ m.lastForwarders in chronological order) {
        Feedback $F'$ := {f | p = f.subject $\wedge$ f $\in$ **this**.FeedbackRepository };
        add $F'$ to F;
        **if** (|F| > b/2) **break**;
    }
    // (2) get feedback objects close to $P_t$
    sort **this**.FeedbackRepository by dist($P_t$, f.subject) with f $\in$ **this**.FeedbackRepository;
    **forall** (f $\in$ **this**.FeedbackRepository $\wedge$ f.subject $\neq P_t$) {
        add f to F;
        **if** (|F| > b) **break**;
    }
    **return** F;
}
```

Fig. 3. Feedback Dissemination

forwarders, until there are $b/2$ feedback objects in the attachment, or the issuer of the message is reached. Regarding (2), the current peer then looks at the peer it intends to forward the message to. It orders the feedback objects in its repository by the distance of the subject and the target peer. It then adds objects from the top of the list to the attachment of the message until its size is b.

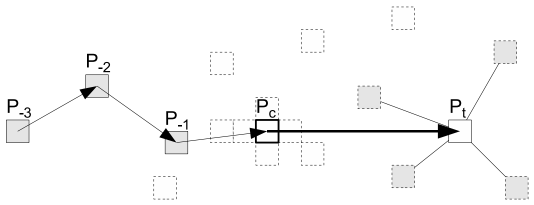

Fig. 4. Forwarding Feedback.

Example 4: Suppose the peer P_c in Figure 4 is about to forward a message to P_t. Peers that have forwarded the message in the past are labeled with P_{-1}, P_{-2}, P_{-3}. Other peers known by P_c are shown as dashed boxes. Assume P_c has feedback available about all peers depicted in the figure. It has to determine b objects to be attached to the message. Following our protocol, P_c will select Feedback objects whose subjects are the nodes in light grey. □

4.4 Managing Local Feedback Objects

Each peer administers a repository of Feedback objects. It must decide which objects should be inserted into or removed from the repository. Some rules for removal are simple – feedback that has expired can be discarded. Insertion is more complex: with the protocol described so far, the number of incoming or newly generated Feedback objects is very large. Thus, when obtaining feedback, a node works off the following rules:

R1 *If a Feedback object is part of a message from a peer that the current peer does not deem reliable, then discard it.*

R2 *If the timestamp of a Feedback object is older than e, then discard it.*

R3 *If the repository contains a Feedback object s.t.*
- *both the originators and the subjects of the incoming Feedback object and the one in the feedback repository are identical, and*
- *the originator of the feedback is different from the current peer,*

then keep the object whose timestamp is newer, and discard the other one.

R4 *If the feedback repository already contains at least t Feedback objects about the same feedback subject, append the incoming one, and remove the Feedback object with the oldest timestamp.*

R1–R3 ensure that the feedback repository contains up-to-date feedback from reliable sources. R3 prevents from perceiving a peer as reliable based on observations of a single node only. An exception is feedback from the current node itself. R4 avoids unnecessarily large numbers of Feedback objects. Since t objects are already sufficient for reliability, more feedback does not provide any further value. Having survived Rules R1–R4, a Feedback object is added to the feedback repository. If the size of the repository exceeds $s_r \times t$, all Feedback objects regarding one peer are removed. That peer is the one with the smallest number of valid Feedback objects. This is natural, because peers can set unreliable peers aside, but want to keep useful ones.

4.5 Reliability-Aware Forwarding

We now explain how peers use feedback information. Our objective is twofold: on the one hand, we want peers to forward messages via reliable peers as far as possible. On the other hand, query results must not be given to peers that might be uncooperative. A peer estimates the reliability of another peer by counting the respective Feedback objects in its feedback repository. If the number is at least threshold t, it is reliable. A valid Feedback object is one that has passed the rules from Subsection 4.4.

Method *forwardMessage* (Figure 5) is responsible for sending messages to appropriate peers. Note that Message m, which is parameter of *forwardMessage*, always contains a key to determine the target of the message in the key space. If the message is not a query, the key is the center of the zone of the target peer. Reliability-aware greedy forwarding now works as follows: each peer wants to find not only a close, but a close *reliable* node that is nearer to the target of the message than itself. If the peer has such a node in its contact cache, it sends it the message. If not, it makes a distinction between query results and other messages.

```
forwardMessage(Message m) {
    // determine candidates to forward the message to
    CandidatePeers C := $\{p \mid dist(p, m.key) < dist(\textbf{this}, m.key) \wedge p \in \textbf{this}.ContactCache\}$;
    sort C by dist(m.key, p) with $p \in C$;

    // search for a reliable addressee
    forall ($p \in C$) {
        Feedback F := $\{f \mid f.subject = p \wedge f.type = positive \wedge f \in \textbf{this}.FeedbackRepository\}$;
        if ($|F| \geq t$) {
            m.attachment = generateFeedbackAttachment(m, p);
            send(m, p);
            return;
        }
    }
    // the current peer does not know a reliable node
    if (m.type = query result) {
        Neighbors N := all neighbors of this in C;
        forall ($p \in N$) {
            requestProW(p);
            waitForProWAnswer (timeout);
            if (ProW answer returned in time) {
                generateFeedback(p);
                m.attachment = generateFeedbackAttachment(m, p);
                send(m, p);
                break;
            }
        }
    } else {
        Peer p := first element in C;
        m.attachment = generateFeedbackAttachment(m, p);
        send(m, p);
    }
}
```

Fig. 5. Reliability-Aware Forwarding in CAN.

Query Results: If the current peer wants to forward a query result, it does so to one of its (possibly unreliable) neighbors in the right direction, but asks for a ProW before doing so. If the ProW arrives in time, it forwards the message to the neighbor. Otherwise, the peer tries another neighbor. In the extreme situation that there is no further contact, the current peer drops the message. [19] tells us that a P2P system must not provide any service to nodes that have not yet proven their willingness to cooperate. Therefore such messages must not go to unknown peers until they have proven their reliability. The ProW is limited to neighbors for security reasons: this prevents peers from asking random other ones for a ProW in order to perform DoS attacks.

At first sight, carrying out ProW in the context of evaluation of queries issued by other peers is not dominant. However, recall our assumption that peers issue queries at a steady rate. A peer with a poor standing would have to carry out a ProW insignificantly later anyhow, when issuing a query itself. Besides that, doing a ProW now does not delay the evaluation of its own query later on. – Further, ProW might seem to be a disincentive to join the network. But the issue is application-specific, i.e., is the benefit from joining the CAN higher than the ProW cost plus the cost of processing queries? Our experiments in Section 5 indicate that the number of ProW is rather small, so the answer to the question should be affirmative in most scenarios.

Other Messages: If a message is not a query result, an uncooperative node cannot benefit from it. So the current peer selects the peer closest to the message key from its contact list and sends it the message. A node that is not reliable in the eye of others is either uncooperative or did not yet have a chance to prove its cooperativeness. The hope is that the second case is true.

4.6 Discussion

So far, we have described a protocol that detects and excludes uncooperative nodes. Let us say why this is the case: Recall that a node is uncooperative if it tries to benefit from the network with little effort. There are two ways to do so: (1) suppress messages, by not answering or not forwarding them, (2) propagate spoof feedback, be it with the peer in question as subject, be it with another peer. Both variants do not result in any benefit: Namely, messages travel via chains of reliable peers, whenever possible. Only messages containing no query results are sent to unknown, but not necessarily uncooperative peers. Integration of peers depends on observations made by reliable peers. Every peer discards incoming feedback from a peer that it does not deems reliable. Since feedback expires after some of time, peers have to keep proving their reliability.

Applicability of the protocol is important as well. Because our protocol bypasses not only uncooperative peers, but also suspicious ones, the number of peers forwarding a message increases. On the other hand, many lost messages have to be repeated in conventional CAN protocols in the presence of free riders. So their costs are not as low as it seems at first sight. Furthermore, peers do not send out messages only to share feedback information with our protocol, and all information attached to a message is strictly limited in size. Finally, logical networks of peers make it unnecessary that a peer stores feedback about all other peers.

Our protocol gives rise to many questions. The effectiveness of our selection policy of Feedback objects to be forwarded is unclear. Next, small contact caches ($s_c < 0.02\%$ of all peers) have turned out to be surprisingly efficient for enhanced CAN [15]. We wonder if small contact caches and feedback repositories, together with a small number of Feedback objects appended, is effective as well. Here, effectiveness means 'good differentiation between cooperative and uncooperative peers, with moderate overhead'. We will now address these questions experimentally.

5 Experimental Evaluation

We have evaluated our reliability-aware CAN protocol by means of extensive experiments. The most important question is as follows: How well does the protocol detect uncooperative behavior (Subsection 5.3)? Put differently, does it pay to be cooperative from the perspective of an individual node? Subsection 5.2 addresses another question: What is the overhead of our reliability-aware CAN protocol, as opposed to the other protocols from this article?

Our cost measure is the number of message hops, i.e., the number of peers involved in a single CAN operation. This is in line with other research on DHT [20]. This article leaves aside characteristics of the physical network, such as total latency. We have an implementation of CAN that is fully operational, and our experiments use it as a platform. All experiments ran on a cluster of 32 loosely coupled PC, equipped with 2 GHz CPU, 2 GB RAM and 100 MBit Ethernet each. [21] provides more information on our experimental framework.

5.1 Determining Parameter Values

For the evaluation, we must come up with meaningful parameter values (cf. Table 1).

$\mathbf{n}$, $\mathbf{d}$*:* The claim behind CAN is web scalability. However, with existing CAN protocols where free riders remain unknown, scalability is bounded. The longer the paths, the larger the probability that a message is lost (cf. Example 2). To verify that this is not the case with our protocol, we have a large number of peers (100,000) and a small dimensionality (2), in order to have long paths. Real applications would use a larger d.

$\mathbf{s_c}$*:* [15] has shown that a contact cache of size 20 is adequate, even for large networks.

$\mathbf{u}$*:* In P2P networks without sanctions against free riders, their number is large [22]. In our setting in turn, cooperative behavior is expected to dominate. Hence, a fraction of 5% of uncooperative peers is a highly conservative value.

$\mathbf{s_r}$, $\mathbf{b}$*:* Feedback objects are small, i.e., a few bytes for the identifiers of feedback subject and object and the timestamp. However, their number should be limited, because processing them is resource-consuming. We estimate that 20 Feedback objects attached to a message and a feedback repository size of 100 are viable, even for mobile devices.

$\mathbf{q}$*:* The number of Feedback objects generated for each forward depends on the application on top of the CAN. We assume that storing data and answering queries is ten times as expensive as forwarding messages. So we set q to 0.1.

$\mathbf{t}$, $\mathbf{e}$*:* Threshold t and lifetime[2] e are security-relevant parameters. A low threshold t and a high value for e allow uncooperative peers to get by with processing only few incoming messages. The opposite case, i.e., large t and small e, would burden cooperative peers with ProW requests. We use values $t = 3$ and $e = 100,000$, obtained from previous experiments (not described here for lack of space).

[2] Here the unit of time is given in experiment clock cycles, i.e., this is the number of queries issued.

5.2 Performance Aspects

We anticipate that the number of hops per message is higher with the reliability-aware CAN protocol, than with the enhanced CAN. The reason is that the target node of a message is not the node that is closest to the destination, but the closest *reliable* node. An experiment examines these issues quantitatively. Next to the number of hops, we are also interested in the number of proofs of work requested. A proof of work[3] also leads to an additional pair of messages.

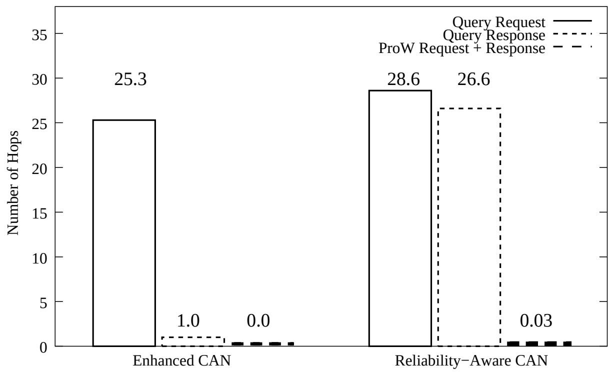

Fig. 6. Total number of hops.

This experiment uses 2,000,000 queries whose keys are uniformly distributed. We also have carried out experiments with real data, but they do not provide any further insight. We omit them here for lack of space. In order to compare the overhead of our protocol with the enhanced CAN running on optimal conditions, we use cooperative nodes only. In the presence of free riders, enhanced CAN would loose many messages, distorting the measurement results. In other words, the setup lets enhanced CAN looks better.

After an initialization period of 500,000 queries, we counted the message hops for each query. We distinguish between message hops necessary to deliver the query itself, those necessary to return the result, and those necessary to request and to return a ProW. Figure 6 graphs the number of hops per query. The figure tells us that the overhead (number of message hops) of the reliability-aware protocol for cooperative nodes is reasonable. Delivering queries only takes slightly more hops in the reliability-aware CAN than in the enhanced CAN. Clearly, our protocol must forward query results between reliable nodes instead of returning them directly to the issuer. So the number of hops is now around twice as large, which we deem acceptable. Finally, the number of proofs of work is tolerable as well.

[3] We are interested in general characteristics of the reliability-aware CAN. To this end, it is sufficient to simulate the proof of work. The advantage is that this does not slow down the experiments.

5.3 Effectiveness

Uncooperative peers try to reach their goals with minimal effort. A peer may try to trick the feedback mechanism by processing only a fraction of incoming messages, hoping that this is sufficient to become cooperative in the eyes of other peers. In what follows, we examine if this kind of uncooperative behavior may be successful. To do so, we have refined the uncooperative peers. First, they never return proofs of work requested. This is natural because these are the most costly requests. Second, they react to some, but not all incoming messages. The percentage of messages reacted to is a parameter that we adjust in our experiments. In what follows, 50 peers are reacting with 0% probability, another 50 with 1% and so on up to 99%. The remaining peers are fully cooperative. Our objective is to block all uncooperative peers, independent of the degree of uncooperativeness, and serve only fully cooperative ones.

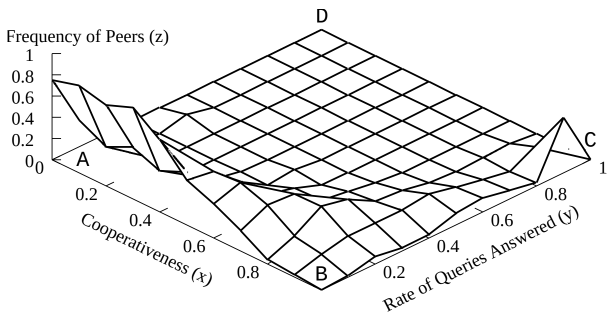

Fig. 7. Behavior vs. Benefits

Each peer corresponds to a point in the xy-plane in Figure 7. The x-coordinate of a peer is the percentage of the messages it reacts to. Its y-coordinate is the rate of its queries that are successful. In other words, a fully cooperative peer that does not obtain any result to its queries corresponds to Point B. A fully uncooperative peer that obtains results to all of its queries corresponds to Point D. There should not be any such peers; and our protocol works well if uncooperative peers have a low rate of 'successful' queries. Note that we can already 'declare success' if a lower degree of cooperation leads to a much lower rate of 'successful' queries *on average*. The reason is that this is sufficient to deter uncooperative behavior.

The z-axis is the number of peers that correspond to the point in the xy-plane. For instance, consider the z-values corresponding to points on the y-axis, i.e., fully uncooperative peers. There are no fully uncooperative peers that benefit much from the CAN, since $z = 0$ for $y > 0.3$. This is a positive result. Analogously, consider the z-values corresponding to points with $x = 1$, i.e., fully cooperative peers. The y-coordinate of most of these points falls into the interval $[0.8; 1.0]$. In other words, cooperative peers have most of their queries answered. This is again positive. Note that the values on the z-axis are scaled to 1 in the direction of peers with the same degree of cooperativeness. That is, the sum of all peers with the same behavior equals 1. This is why the elevation at the bottom left of the figure is very high. Finally, the figure tells us that the CAN more or less blocks all peers that are uncooperative and serves only cooperative ones. This is in line with our objective mentioned above. Our main result is that the protocol levels up to our objectives. Cooperative behavior actually pays off.

6 Related Work

Distributed Hash Tables administer a large number of (key, value)-pairs in a distributed manner, with high scalability. The variants, next to CAN, differ primarily in the topology of the key space. *LH** [23] determines nodes responsible for a certain key statically by its hash function. *Chord* [12] organizes the data in a circular one-dimensional data 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. *Pastry*, *Tapestry* [13, 14] use a Plaxton Mesh to store and locate its data. The forwarding algorithm is similar to the one of Chord. Pastry and Tapestry forward to peers such that the common prefix of the ID and the query key becomes longer with each hop. Because of the organization as a Plaxton Mesh, multiple routes to the addressed position in the data space are possible. With CAN in turn, the number of possible alternative routes for forwarding messages increases with the number of neighbors, i.e., with the dimensionality of the key space. The choice of possible paths is much bigger with CAN. This is important to bypass unreliable peers.

Free riding [2] is an important problem in any computing environment with many anonymous participants. There are approaches against free riding in many different application scenarios. Related work in mobile ad-hoc networks [4, 5] assumes that adjacent nodes can eavesdrop traffic in the same radio network cell and control access to the parts of the network they are supposed to forward messages to. Here, detecting and punishing uncooperative behavior is easy.

[10] uses a P2P network to run a global reputation repository. The approach does not address most of the questions that are relevant in our context, e.g.: Who should be allowed to give feedback? What to do with feedback that comes from untrusted peers? What happens if the originator of a feedback item becomes malicious? From a different perspective, our contribution is a tight coupling of reliability management and message forwarding in P2P networks. [10] in turn deals with trust management on top of such a network.

Another approach to rule out uncooperative behavior is based on micropayments [9, 24]. But while monetary schemes provide a clean economic model, infrastructure costs may simply be too high in a setting such as ours. A further disadvantage is that they require a central bank. This is not in line with our design rationale.

[11] offers a direct way of sharing reputation information without intermediates. Every peer describes each other node with a rating coefficient, i.e., a numeric value. The coefficients are shared after every transaction between nodes involved. A node updates its coefficients by adding the new value weighted by the coefficient of the sender. This is not applicable to large networks because the way of updating coefficients limits reputational information to nodes next to the rated node.

Banning uncooperative nodes may also be the result of using Public Key Cryptography. Public Key Certificates signed by a large number of peers provide verifiable identities [8]. The idea is that groups of peers are mutually verifying and signing their identities. Unfortunately, whenever such a group recognizes that one of their members became uncooperative, certificates must be revoked. In other words, an individual uncooperative peer may break its entire group.

7 Conclusions

This article has presented a CAN protocol that deals with one of the biggest obstacles in P2P systems, namely free riding. In CAN, uncooperative peers basically are those that do not process incoming messages related to queries issued by other nodes. Our protocol explicitly acknowledges work carried out by peers. This facilitates the emergence of self-organized virtual networks within the CAN. The protocol ensures that unreliable peers do not obtain any benefits. Uncooperative behavior is unattractive. The 'downside' of the protocol are slightly longer message paths, in order to bypass unreliable peers, and a number of proofs of work that seemingly unreliable nodes must perform. Several issues remain open for future research. We for our part want to address security issues.

References

1. Ratnasamy et al., S.: A Scalable Content-Addressable Network. In: Proceedings of the ACM SIGCOMM 2001 Conference, New York, ACM Press (2001) 161–172
2. Ramaswamy, L., Liu, L.: Free riding: A new challenge to peer-to-peer file sharing systems (2003)
3. Axelrod, R.: The Evolution of Cooperation. Basic Books, New York (1984)
4. 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, EPFL-IC-LCA CH-1015 Lausanne, Switzerland (2003)
5. Buttyan, L., Hubaux, J.P.: Enforcing Service Availability in Mobile Ad-Hoc WANs. In: Proceedings of the 1st ACM International Symposium on Mobile Ad Hoc Networking & Computing, IEEE Press (2000) 87–96
6. Marti et al., S.: Mitigating Routing Misbehavior in Mobile Ad Hoc Networks. In: Mobile Computing and Networking. (2000) 255–265
7. Srinivasan et al., V.: Cooperation in Wireless Ad Hoc Networks. In: Proceedings of the IEEE INFOCOM. (2003)
8. Gokhale, S., Dasgupta, P.: Distributed Authentication for Peer-to-Peer Networks. Workshop on Security and Assurance in Ad hoc Networks. (2003)
9. Golle et al., P.: Incentives for Sharing in Peer-to-Peer Networks. LNCS **2232** (2001) 75ff.
10. Aberer, K., Despotovic, Z.: Managing Trust in a Peer-2-Peer Information System. In: Proceedings of the CIKM-01, New York (2001) 310–317
11. Padovan et al., B.: A Prototype for an Agent based Secure Electronic Marketplace Including Reputation Tracking Mechanisms. In: HICSS. (2001)
12. Stoica et al., I.: Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In: Proceedings of the ACM SIGCOMM 2001 Conference. (2001) 149–160

13. 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) 329–350
14. Zhao, B.Y., Kubiatowicz, J., Joseph, A.D.: Tapestry: an infrastructure for fault-resilient wide-area location and routing. Technical Report UCB//CSD-01-1141, University of California at Berkeley (2001)
15. Buchmann, E., Böhm, K.: Efficient Routing in Distributed Scalable Data Structures (in German). In: Proceedings of the 10th Conference on Database Systems for Business, Technology and Web. (2003)
16. Back, A.: Hashcash - A Denial of Service Counter-Measure. http://www.cypherspace.org/∼adam/hashcash/ (2002)
17. Jakobsson, M., Juels, A.: Proofs of work and bread pudding protocols. In: In Proceedings of the IFIP TC6 and TC11 Joint Working Conference on Communications and Multimedia Security (CMS ’99), Leuven, Belgium. (1999)
18. Xiong, L., Liu, L.: A Reputation-Based Trust Model For Peer-To-Peer Ecommerce Communities. In: IEEE Conference on E-Commerce (CEC’03). (2003)
19. Friedman, E., Resnick, P.: The Social Cost of Cheap Pseudonyms. Journal of Economics and Management Strategy **10** (1998) 173–199
20. Kleinberg, J.: The Small-World Phenomenon: An Algorithmic Perspective. In: Proceedings of the 32nd ACM Symposium on Theory of Computing. (2000)
21. Buchmann, E., Böhm, K.: How to Run Experiments with Large Peer-to-Peer Data Structures. In: Proceedings of the 18th International Parallel and Distributed Processing Symposium, Santa Fe, USA. (2004)
22. Adar, E., Huberman, B.: Free Riding on Gnutella. First Monday **5** (2000)
23. Litwin, W., Neimat, M.A., Schneider, D.A.: LH* - Linear Hashing for Distributed Files. In Buneman, P., Jajodia, S., eds.: Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993, ACM Press (1993) 327–336
24. Yang, B., Garcia-Molina, H.: PPay: micropayments for peer-to-peer systems. In Atluri, V., Liu, P., eds.: Proceedings of the 10th ACM Conference on Computer and Communication Security (CCS-03), New York, ACM Press (2003) 300–310