Trade-offs Between Reliability and Overheads in Peer-to-Peer Reputation Tracking

Minaxi Gupta[a]*, Mostafa H. Ammar[b], Mustaque Ahamad[b]

[a]Department of Computer Science, Indiana University,
Bloomington, IN 47405, USA

[b]College of Computing, Georgia Tech,
Atlanta, GA 30332, USA

Keywords: peer-to-peer, Gnutella, reputations.

1. Introduction

The peer-to-peer (P2P) paradigm allows participants to exchange resources like content, processing power, and storage capacity without the need for a centralized authority. In particular, the focus of this paper are the highly popular file-sharing P2P networks like Gnutella and Kazaa that specialize in decentralized content distribution. Each peer in these networks is capable of assuming the role of a client as well as a server. Peers form application layer overlays in order to search for content and download it from among the available choices.

The deployed instances of P2P networks implicitly possess several unique features. They are essentially *goodwill* networks of mostly *unknown* peers and the peer population in these networks tends to be highly *transient*. The result is that the deployed instances of these networks suffer from issues like *freeriding* (Freeriders are peers that download content from others but do not want to serve it to other peers) and *lack of trust*.

Research in mechanisms to counter freeriding and promoting trust to enhance cooperation in P2P networks has gained momentum. Using evolutionary prisoner's dilemma, the authors in [1] have shown that basing the decision to cooperate with a peer on its past behavior (and hence reputation) causes a P2P network to converge to optimal cooperation. Similarly, work in [2] argues that providing different types of service to peers based on their reputations can motivate peers to cooperate in a P2P network.

The ability to track peer reputations is a requirement of all the above incentive techniques. Reputations are a useful feature of any P2P network because they can not only be used in devising incentive techniques but can also be used in making decisions about who to serve content to and who to download content from.

Existing proposals to track peer reputations can be broadly divided into two categories: 1) *subjective* (examples include Binary trust model [3], Xrep [4], EigenRep [5], NICE [6], and Peertrust [7]) and 2) *objective* (examples include Kazaa [8], DCRC [9], and KARMA [10]). In subjective reputations peers maintain personal histories based on their perception of transactions with other peers. This allows for decentralized storage of reputations without the need for special infrastructure but has implications on *reputation inference*. Specifically, subjective reputations incur extra communication overhead when a peer wants to infer the reputation of any other peer in the network. If the peer churn rate is high the inferred reputations may contain *unknown* amount of inaccuracy. Further, since the reputation computations in subjective reputations hinge on the *cooperation* of peers for both storage and computation of reputations, they are subject to peer manipulation. In fact, most subjective rep-

*This work was done when the author was a Ph.D. student at Georgia Tech.

utation tracking schemes implicitly work under the assumption that only a small percentage of peers are uncooperative. The objective reputation schemes differ from subjective schemes in that they use well defined set of criteria to track peer reputations in terms of resource consumption and contribution. However, they require *infrastructure* support for tracking reputations reliably to avoid the problems faced by the subjective schemes.

The issue of reliability in the existing reputation tracking schemes has only been addressed in an ad-hoc manner. A recent work ([11]) uses game theoretic approach for analyzing reputation inference protocols and is a first step in formalizing the reliability aspects in reputation tracking. The overheads in reputation tracking have also not received any attention so far. Understanding the trade-offs between reliability and overheads is crucial in designing viable reputation tracking schemes that do not burden the underlying P2P network. The primary goal of this paper is to enhance the understanding of these trade-offs and explore the design space of reputation tracking schemes in terms of the reliability and overheads trade-offs they offer. Due to the several desirable properties that objective reputation tracking schemes possess that make them more suitable for applications like P2P service differentiation ([2]), we use the objective reputation tracking criteria described in [9] as the substrate for designing reputation tracking schemes that vary in overheads and reliability.

Specifically, we present two solutions for ensuring reliability in objective reputation tracking that vary in the overheads they incur. The first solution, *strong reputations*, offers reliable reputation tracking by defeating attacks from *selfish*[2] peers but incurs high overheads and introduces a centralized bottleneck during content download. We *formally specify* and *verify* the reliability properties of strong reputations. The second solution, *weak reputations*, offers *limited* reliability but has smaller overheads and may offer a more viable alternative for certain applications. We also quantify and compare the overheads involved in each of the proposed solutions.

The rest of this paper is organized as follows. In section 2, we outline the objective criteria described in [9] for tracking peer reputations. Sections 3 and 4 describe the goals and assumptions for strong and weak reputations. Strong reputations and their security properties are described in sections 5 and 6 respectively. The corresponding sections for weak reputations are sections 7 and 8. A comparison of overheads for strong and weak reputations is presented in section 9. Finally, sections 10 and 11 discuss deployment considerations and present the conclusion.

2. An Objective Reputation System

The success of the search phase in Gnutella-like P2P networks requires that the other peers be online, agree to search for the content from their shared directory, and forward the query further depending on the hop count of the query. The success of the download phase requires that the chosen peer be online and serve the content when requested. For successful content retrieval, the type, quality, and quantity of the content each peer places in the shared directory plays an important role.

Guided by the above factors that affect the experience of a peer in a P2P network, the work in [9] describes two methods of computing reputations objectively. Due to the security properties afforded by the debit-credit reputation computation (DCRC) criteria, we focus only on the DCRC scheme as a substrate in designing strong and weak reputations in this paper[3].

The basic idea behind the DCRC scheme is to credit the peer reputation scores for resource contribution and offer debits for consumption of resources. DCRC uses two tunable system param-

[2] Security threats to reputation tracking arise both from *malicious* and *selfish* peers. We focus only on selfish peers in this paper because attacks like denial-of-service (DoS) that the malicious peers can launch are not specific to how reputations are computed and can be launched on any P2P network. As a result, unless the underlying P2P network can be secured from malicious peers addressing maliciousness in reputation tracking is not very useful.

[3] The DCRC outlined here is a variant of the original scheme. Because it is hard to measure the bandwidth a peer uses for serving content, we have eliminated bandwidth from the factors used in reputation tracking.

eters: 1) *file size factor f* and 2) *time factor (in hours) t*. The file size factor, f, determines how many MBytes of data transfer results in a unit increment to the reputation score and the time factor, t, is used to determine the granularity at which peer cooperation in the system for sharing and staying online is rewarded. These parameters ensure that the reputation scores stay within a certain range of non-negative values. They do so by tuning the system for larger file sizes, and longer stay in the system. Utilizing these parameters, a peer's total reputation score is computed using the following four components:

Query-Response Credit (QRC): The DCRC scheme uses average query-response message size to credit peer reputations for processing the query-response messages. If the average query response message size is denoted by QR, the number of points earned for each query processed are given by:

$$QR$$

It has been reported [12] that the average query-response traffic in a Gnutella network is about .75 KB per second per connection. Also, most connections generated about 15 messages per second. This gives a rough estimate of the average query-response message size to be 0.00006 MBytes. This value can be used to specify QR.

Upload Credit (UC) and Download Debit (DD): Peers get credit for serving content and debit for downloading it. For a file of size s MBytes the debit and credit to the reputation score is given by:

$$\frac{s}{f}.$$

Sharing Credit (SC): During the content search phase, in addition to peer availability, another important factor is the amount of content shared. Some peers may be sharing *hard-to-find* content. QRC, UC, and DD may not give any credit to such peers because by definition, such content may not be heavily accessed. SC is intended to capture this effect. In practice, it is very hard to implement this correctly in a lightweight fashion and in fact for the rest of this paper we assume SC is not provided. But assuming it can be implemented, if a peer shares n files where the size of jth file is given by s_j, at the elapse of each time factor the number of points it will earn are given by:

$$\sum_{j=1}^{n} \frac{s_j}{f}$$

The total reputation score RS for a peer k who processes a query-response messages, facilitates b uploads, performs c downloads in d time factors is given by:

$$RS_k = a \times QRC + \sum_{l=1}^{b} UC_l - \sum_{m=1}^{c} DD_m + d \times SC$$

where UC_l and DD_m are the upload credit and download debit for files l and m respectively.

The authors in [9] recognize the reliability implications of allowing the peers to update their own reputation scores using the above objective criteria. They propose utilizing an infrastructure of *reputation computation agents* (RCA) for fair periodic updates to each enrolled peer's reputation using the above criteria, still ensuring that the reputation points for each peer are kept locally for fast retrieval.

3. Design Space and Reliability Goals

The security aspects of computing reputations reliably in the DCRC scheme were only addressed in an ad-hoc manner in [9]. In this paper, we explore the spectrum of solutions to compute reputations objectively that utilize the RCA infrastructure and vary in overheads and reliability. In particular, we propose two specific reputation computation schemes with varying overheads on the underlying P2P network and analyze the extent of reliability and possible security attacks. It is noteworthy that even though the proposed solutions appear to resemble the e-commerce and e-cash schemes there are some important differences. First, the available e-commerce and e-cash schemes are applicable only for a single payer and payee pair while in the P2P context a single file download is facilitated by many peers. Secondly, though some e-commerce and e-cash schemes provide weaker guarantees than others no one scheme

allows tuning of trade-offs between overheads and security guarantees.

Figure 1 shows the design space of solutions. The leftmost end of the spectrum corresponds to the minimum overhead solution. The current implementations of Gnutella like P2P networks which do not have any support for reputation tracking fall here. The rightmost end of the spectrum corresponds to a hypothetical fool-proof reputation tracking solution which is capable of guarding against all possible security attacks[4]. As expected, this is the maximum overhead solution. Any solution that tracks reputations falls in between these two ends of the spectrum.

Kazaa uses the notion of *participation level* ([8]) in order to track peer contribution. Kazaa software locally updates the net MBytes served by each peer. In times when the request rate is high, the participation level is used to prioritize among downloaders. Though this solution has very little overhead, it offers no security against selfish peers that know how to alter the part of their software that computes participation level.

The strong and weak reputations proposed in this paper in sections 5 and 7 fall in between Kazaa's participation level and the hypothetical fool-proof reputation tracking solutions in terms of overheads and reliability. Both the solutions utilize the RCA infrastructure to varying degrees in offering reliability to the reputation computations. The weak reputations can conceptually be thought of as similar to the reputations maintained in eBay in terms of reliability. Buyers and sellers in eBay can leave incorrect feedback and can also collude. However, the reputations so maintained are still useful. Specifically, for peers that behave in self-interest, we prove that the strong reputations satisfy the following properties:

1. *only peers that contribute to the search and download should be able to collect credit.*

2. *no downloading peer should be debited unless it receives the content completely.*

3. *no serving peer should be able to collect credit without serving the content completely.*

4. *the credit and debit settlement should occur at most once for each content download.*

[4] We call it hypothetical because it is resistant against even undiscovered attacks.

We divide collusion scenarios into two types: 1) those where peers acting out of self-interest collude to increase each other's reputation scores and 2) those where the increase in one peer's reputation score results in a penalty to the reputation score of the peer facilitating it. The above properties are only valid for the first scenario. The peers contributing to the second type of collusion do not act out of self-interest but do not fall under the category of malicious peers either. Section 10 discusses the implications of the second type of collusion.

4. Assumptions

We begin by discussing the assumptions about the functionality of the RCA infrastructure (section 4.1) and the terminology used (section 4.2) before describing the details of strong and weak reputations.

4.1. Infrastructure

For simplicity of the subsequent description, we assume that the RCA infrastructure is a single entity. Issues in the division of functionality among various RCAs are discussed in section 10. The RCA is also assumed to be a trusted entity in the system but peers can collude with other peers in self-interest.

A peer who is interested in getting its reputation tracked first enrolls itself with the RCA to get a (public, private) key pair. This distribution of (public, private) key pair can be thought of as similar to that of public key infrastructure (PKI) in the sense that it permits only one *enrolled* identity per peer[5]. Alternately, a scalable public key distribution infrastructure such as the one proposed in [13] can also be used, if one is available.

[5] A peer can however generate any number of *unenrolled* identities but they can not be used in earning a good reputation score.

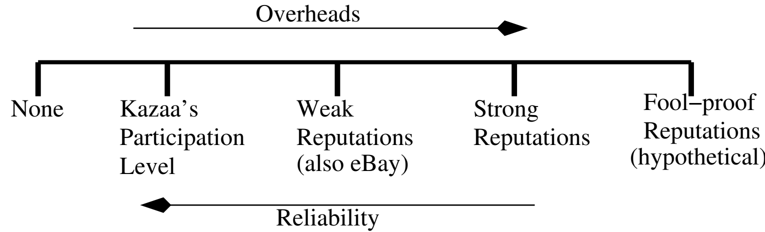


Figure 1. Spectrum of reliable reputation computation solutions.

The digest of a peer's public key is used to identify it. We assume that the (public, private) key pair of the RCA is denoted by $\{PK_{RCA}, SK_{RCA}\}$ and that each peer has access to the RCA's public key and that the peers can obtain public keys of other peers in the system when needed.

For strong reputations, the RCA is expected to have a copy of all the content served by the enrolled peers for the purposes of ensuring content reliability[6]. The creators of the content can provide the RCA with the content when they initially create it. Note that this requirement only incurs a storage overhead on the RCA. Since the RCA is not required to serve this content there is no serving overhead. Further, the RCA is not required to store content in the case of weak reputations. The strong reputations also require the RCA to periodically crawl the P2P topology to ensure only valid QRCs are processed[7].

For privacy or other reasons, peers may not want to get their reputations tracked. As a result, enrollment in the reputation computations is *voluntary*. Peers who choose not to enroll always maintain a default reputation score of 0, the minimum allowable by the system. Peers who enroll can enhance their scores by being good citizens of the P2P network. They can also save their reputation scores across sessions. Thus, a cooperative peer can maintain benefits of its participation in the system in spite of being off-line for a while.

[6]We assume all the content shared is legal.

[7]Offering QRC may be hard in real systems because the crawl overheads may be prohibitive in the presence of high peer churn rates in large P2P systems.

4.2. Terminology

A P2P system comprises of only one type of entity - the peers. All the peers in a P2P system are treated homogeneously in spite of the heterogeneity. Due to the introduction of voluntary enrollment and the RCA a P2P system now has three types of entities: 1) enrolled peers, 2) unenrolled peers, and 3) RCA.

Depending on the stage of the content retrieval and the role of an enrolled peer, its designation changes. We refer to the peer that generates the query as the *querying peer/querier* in this paper. All the peers that receive and process the query are called *searching peers/searchers*. Once the querying peer receives all the replies, it chooses a peer to download the content object from. At that point, it becomes a *downloading peer/downloader*. The peer that serves the content is referred to as the *serving peer/server*. The (public, private keys of querier, searcher, downloader, and server are denoted by $\{PK_{query}, SK_{query}\}$, $\{PK_{search}, SK_{search}\}$, $\{PK_{download}, SK_{download}\}$, and $\{PK_{serve}, SK_{serve}\}$ respectively.

5. Strong Reputations

The reputation tracking in strong reputations is essentially done in two steps: 1) searchers save proofs of their contributions during content search to collect query-response credit (QRC) and the RCA saves *transaction state* about content download that can be converted into upload credit (UC) for the servers and 2) searchers periodically send QRCs to the RCA. The RCA pro-

cesses the QRC for the searchers, UC for the servers, and the corresponding download debit (DD) for the downloaders. It then sends the encrypted QRCs and UCs as reputations to each for keeping locally but stores the DDs with itself to ensure they are not dropped by the recipient peers.

5.1. Proofs of Peer Contributions

pSearch: To save the proofs of their contributions, for every query-response message processed during content search the enrolled searchers save {*searcher_identity*, *querier_identity*, *query_keywords*, *time_stamp*} $_{SK_{query}}$ as the *proof of searching (pSearch).*

pServe: The RCA facilitates content download for reliability of content delivery. The following protocol takes place between an enrolled downloader and enrolled server at the time of the file download[8]:

Step 1: The downloader sends a request for content to the server, denoted by *DownloadReq.* This message contains {*downloader_identity, content_identity, time_stamp*} $_{\mathrm{SK}_{\mathrm{download}}}$.

Step 2: The server generates a session specific symmetric key called *SymKey* using the information in *DownloadReq* and encrypts the content with it. This encrypted content is then sent to the downloader as *ContentDelivery.* Encrypting the content guarantees that the downloader is not able to access the content unless it has access to the SymKey.

Step 3: The server sends *KeyTransfer* message which contains {{*server_identity, downloader_identity, content_identity, time_stamp*} $_{\mathrm{SK}_{\mathrm{serve}}}$, *SymKey*} $_{\mathrm{PK}_{\mathrm{RCA}}}$ to the RCA. Encryption by PK_{RCA} ensures no one but the RCA is able to decrypt the information and signing by SK_{serve} ensures that the identity of the server is verified.

Step 4: After receiving the encrypted content, the downloader sends *ContentRcvd* message containing {*server_identity, downloader_identity, content_identity, encrypted_content_digest, time_stamp*} $_{SK_{\mathrm{download}}}$. This message confirms that the downloader received the encrypted content. The *time_stamp* in this message indicates the time at which the downloader received the encrypted content. The digest is useful in verifying the integrity of the encrypted content and can be produced by using an algorithm like MD5 [14] or SHA-1 [15].

Step 5: After decrypting the *ContentRcvd* message from the downloader, the RCA first encrypts the content locally stored under *content_identity* (using SymKey from the *KeyTransfer* message) and calculates its digest. The matching of *content_identity* ensures that the downloader is not tricked into receiving any other content than the one it selected to get. We assume that the RCA uses the same algorithm to compute the digest as the downloader. If the digests match, the integrity of the content is verified.

To complete the transaction between the serving and requesting peers, the RCA can now pass the SymKey key securely to the downloader. It sends a *KeyDelivery* message with {*content_identity, RCA_ID, SymKey*} $_{\mathrm{PK}_{\mathrm{download}}}$ to the downloader. Encrypting this message with downloader's public key ensures that the message can not be snooped upon and that only the downloader is able to decrypt it.

The RCA does not store the SymKey with itself after delivery. However, it does store *transaction state* of the form (*downloader_identity*, *server_identity*, *file_name*, *file_size*, *time_stamp*, *credit_processed_list*) for reputation credit inference purposes. The credit_processed_list is the list of peers who have already received the credit. It could have multiple entries for each peer depending on what type of credit it has already received. For example, the list could have one entry each for QRC, UC, and DD for each peer. The RCA uses this list to ensure that peers receive each type of credit only once.

Figure 2 shows the communication between the RCA, server, and the downloader in the above protocol.

5.2. Processing at the RCA

Periodically, the enrolled queriers send the pSearchs collected thus far to the RCA. Using

[8]It is important to note that if *any* of the peers are not enrolled, the following exchange does not take place. This means that by creating additional *unenrolled* identities peers cannot earn any credit to their reputations.

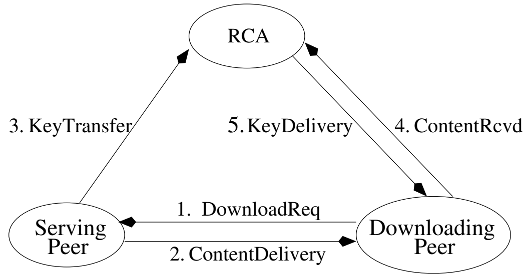


Figure 2. Protocol for secure content download.

the pSearchs and the transaction state, the RCA processes the QRCs, UCs, and the DDs.

Processing upload credit (UC) and download debit (DD): Using the criteria prescribed in section 2, the RCA infers the total UC for the servers and the corresponding DD for the downloaders. It then updates the credit_processed_list for the relevant entries of the transaction state with the identities of the peers that received UC and DD and sends a signed reputation score of the form $\{$*RCA_identity*, *time_stamp*, *reputation_score*, *server_identity*$\}_{SK_{RCA}}$ to the serving peers. The signature allows the peers to store their own reputations locally without being able to tamper with them. To avoid having the downloaders drop negative reputation scores, the RCA retains the DDs in the form of *debit state* with itself until those peers send some UCs for processing. Upon receiving the UCs the RCA first applies the DDs and sends a signed reputation score only if the result is positive.

Processing query-response credit (QRC): Since peers can forge QRCs for the files that were never downloaded, the QRCs from the pSearchs are not processed until the corresponding UCs and DDs are processed, irrespective of when they arrive. Also, the RCA uses the topology snapshots it maintains while processing pSearchs. After inferring the QRCs using pSearchs and the transaction state, the RCA updates the credit_processed_list and sends a signed reputation score of the form $\{$*RCA_identity*, *time_stamp*, *reputation_score*, *searcher_identity*$\}_{SK_{RCA}}$ to the peers that sent the QRCs.

5.3. Formal Specification of Strong Reputations

The formal specification of strong reputations using a version of *Abstract Protocol Notation* [16] is presented in the APPENDIX. It contains a process each for the 5 entities: *queriers*, *searchers*, *downloaders*, *servers*, and *RCAs*. We assume that there are n peers and m distinct files in the system and that the encryption and decryption are denoted by NCR and DCR respectively.

6. Attack Analysis for Strong Reputations

For peers that behave in self-interest, we prove that the strong reputations satisfy the following properties:

1. *only peers that contribute to the search and download should be able to collect credit.*

2. *no downloading peer should be debited unless it receives the content completely.*

3. *no serving peer should be able to collect credit without serving the content completely.*

4. *the credit and debit settlement should occur at most once for each content download.*

6.1. Attacks and Actions

To defeat the above security goals an adversary can mount the following attacks to compromise the reliability of reputation computations:

- **a0:** earn credit without doing any work in the system.
- **a1:** earn credit for useless work.
- **a2:** earn credit for partial file transfer.
- **a3:** earn credit instead of someone else (with or without their consent).
- **a4:** earn credit more than once for the same work[9].

[9]Since every credit has a debit, earning credit more than once implies some other peer receives undue debit.

- **a5:** get content without earning a debit.

To launch each of the above attacks, a selfish peer can use one or more *actions*. These actions can broadly be divided into 3 categories: 1) *identity related*, including *impersonation* and *repudiation*, 2) *IP address related*, including *spoofing* and *TCP hijacking*, and 3) *message* and *content related*. Out of these three categories of actions, only the last one can be used to mount attacks on strong reputations. The identity related actions are ruled out because each enrolled peer is uniquely identifiable by the digest of its public key and the process of (public, private) distribution is assumed to be safe. The IP address related actions are not of concern because a peer is identified only through the digest of its public key and is allowed to legitimately change IP addresses while maintaining the same identity. We next formalize the definitions of possible message and content related *actions* that can be used by selfish peers to launch attacks on strong reputations:

Forgery: Generation of fake messages or content by an enrolled peer in collusion with, or without another enrolled peer.

Modification: Alteration of messages or content in transit through interception.

Replay: Retransmission of a valid message either by the originator or by another peer who snooped on the message in transit.

6.2. Formal Verification

Figure 3 shows the flow of messages in strong reputations between the *querier (q)*, *searcher (sc)*, *downloader (d)*, *server (sr)*, and the *RCA (r)*. The content search and download related messages are shown by solid lines and the debit-credit settlement messages are shown by dashed lines.

The interactions among the five entities in figure 3 can be divided in three main categories: 1) between the querier q, searcher sc, and the RCA during content search and the corresponding credit settlement (denoted by $q-sc-r$), 2) between the downloader d, server sr, and the RCA r during content download (denoted by $d-sr-r$), and 3) between the RCA r, and the server sr during debit-credit settlement (denoted $r-sr$).

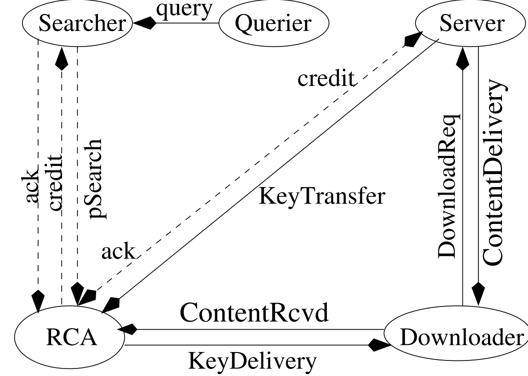


Figure 3. Interactions among the five entities.

Figure 1 shows a listing of possible attacks during each of the three interactions.

Table 1
Possible attacks during interactions.

	$a0$	$a1$	$a2$	$a3$	$a4$	$a5$
$q-sc-r$	✓			✓	✓	
$d-sr-r$		✓	✓	✓	✓	✓
$r-sr$	✓			✓		

We use *convergence theory* to verify the reliability of strong reputations. In convergence theory, a protocol is called *secure* if it satisfies the following three conditions [17]:

Closure: In each protocol computation whose first state is *safe*[10], every state is safe.

Convergence: In each protocol computation whose first state is *unsafe*[11], there exists a safe state.

[10]A computation of a protocol is an infinite sequence $(p0, p1, ...)$ of protocol states such that each pair $(pi, p(i+1))$ of successive states in the sequence is a protocol transition. A state is *safe* if it occurs in any protocol computation $(p0, p1, ...)$ where $p0$ is an initial state of the protocol.

[11]A state is *unsafe* if it can be reached by some adversary action starting from a safe state, or if it occurs in some protocol computation $(p0, p1, ...)$ where p0 is an error state of the protocol.

Protection: In each protocol transition whose first state is unsafe, the critical variables of the protocol do not change their values.

As argued in [17] each protocol satisfies the closure condition. This is because the protocol *states* are defined from a valid domain of values and the *transitions* occur only when actions whose guards are true are executed. Hence, to formally prove a protocol is secure, it is sufficient to show that the protocol satisfies the convergence and protection conditions.

For the three categories of interactions, we now describe how the attacks described in figure 1 are launched using forgery (F), modification (M), and replay (R) and how the security built in strong reputations counters them. In the subsequent discussion and in figure 4, the safe states are denoted by Si, $i \in \mathcal{I}$, with S_0 being the initial state for each interaction. The unsafe states are denoted by Ui, $i \in \mathcal{I}$ and the actions of forgery, modification, and replay are labeled as F, M, and R respectively.

6.2.1. Interaction q-sc-r

Attacks $a0$, $a3$, and $a4$ can be launched by selfish peers during credit settlement for content search contribution.

The protocol starts in state $S0$ when a searcher sends out a query in search for content. During normal operation of the protocol, the enrolled searchers save pSearchs as a proof of their work, leading the protocol to move to $S1$. When these pSearchs are sent to the RCA, the protocol moves to state $S2$. Upon receiving the pSearchs, the RCA infers the query-response credit (QRC) and sends them to the searchers, leading the protocol to state $S3$. Finally, when the searchers send acks for receiving QRC, the protocol returns to state $S0$.

From $S0$, the protocol moves to unsafe state $U0$ instead if selfish enrolled peers save forged pSearchs and launch attack $a0$. There are several possibilities for forging pSearchs: 1) peers can guess the key words, time stamps, and the entities involved in valid queries in the system (given that the number of files and peers in a P2P network are likely to be large, forging legitimate transactions and their time stamps is not possible), 2) they can snoop on queries not encountered in their section of the network, or 3) they can obtain information about valid queries from other colluding peers. When the forged pSearchs are sent to the RCA, unsafe state $U1$ results. However, since the RCA maintains snapshots of connectivity information, the forged pSearchs fail to earn any credit and the protocol returns to the initial state $S0$. In state $U1$ however, if a searcher sends pSearchs, unsafe state $U2$ results. The RCA can identify a valid pSearch from a forged one using the topology snapshots it maintains through periodic crawling. As a result, effectively the protocol is in safe state $S2$ and the normal operation of sending the corresponding QRC can be carried out, defeating attack $a0$.

In state $S1$, selfish peers can launch attacks $a3$ and $a4$ by modifying valid pSearchs on the way to the RCA and by replaying old pSearchs respectively. Either of these actions cause the protocol to move to unsafe state $U3$. The transaction state maintained by the RCA helps avoid replays and the signing of pSearchs rules out the possibility of modification. The result is that the protocol returns to safe state $S1$, defeating the attacks. If however, in $U3$, a valid pSearch arrives at the RCA, the protocol moves to unsafe state $U2$. As the modified and replayed pSearchs fail to fetch credit,the net effect is that the protocol returns to state $S2$ and the valid credits can now be processed.

Attack $a3$ can also be launched in state $S2$ when QRCs are being sent out. An enrolled selfish peer can modify a QRC destined for a searcher leading the protocol to an unsafe state $U4$. Since the QRCs are signed, modified QRCs are not of any use. Also, not receiving an ack for the QRC causes the RCA to timeout and the protocol resumes in state $S2$. It is important to notice that QRCs can also be snooped upon in transit. However, that does not cause any interruption to the protocol operation.

6.2.2. Interaction d-sr-r

The interaction between the downloader, server, and the RCA is susceptible to attacks $a1$ through $a5$.

The interaction during content download starts

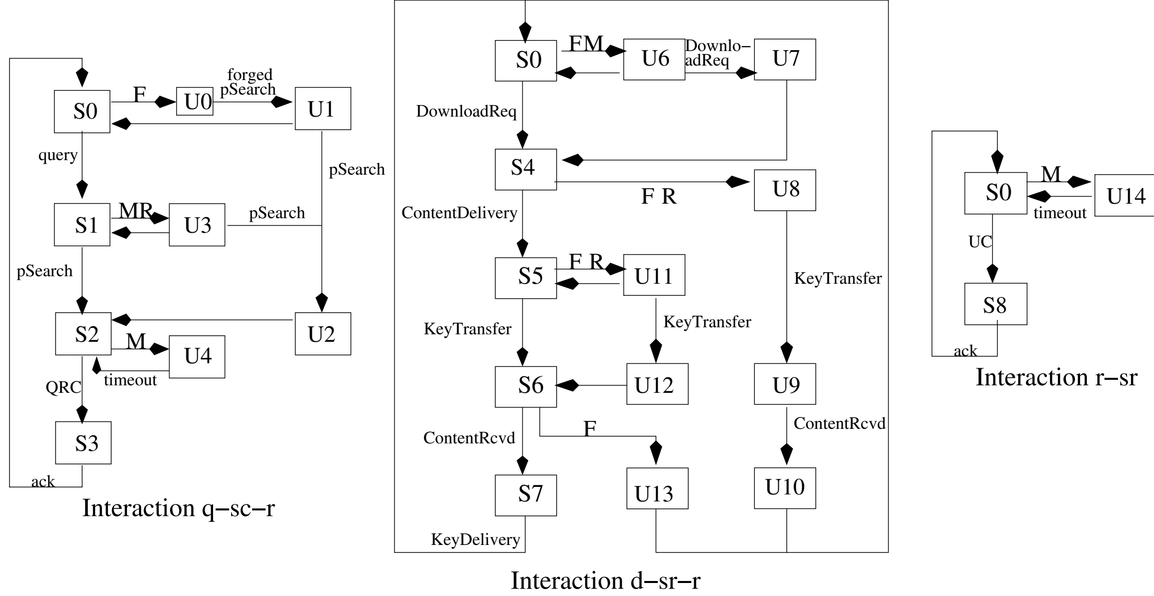


Figure 4. State transition diagrams

in state $S0$. When a downloader sends a *DownloadReq* to the server, the protocol progresses to state $S4$. Upon *ContentDelivery*, the state changes to $S5$. The *KeyTransfer* message leads the protocol to state $S6$. When the downloader sends *ContentRcvd* message, state $S7$ results. Finally, when the RCA sends the *KeyDelivery* message, the protocol returns to its original state.

To launch attack $a5$ a selfish enrolled peer can modify the *DownloadReq* or a selfish searcher could forge an incorrect *DownloadReq* to hide its identity to avoid debit. These actions lead the protocol to unsafe state $U6$. However, since *DownloadReq* is signed, the attack is avoided and the protocol returns to safe state $S0$. If however, a valid *DownloadReq* arrives in state $U6$, unsafe state $U7$ results. The RCA identifies such a request and safe state $S4$ results.

While in state $S4$, a selfish server can launch attacks $a1$ and $a2$ by delivering incorrect or partial content through actions F and R and cause the protocol to reach unsafe state $U8$. Though the server may not be able to save itself much effort, it may do so in the event it does not possess the actual content the downloader selected to get from it. In $U8$, the selfish server can send the *KeyTransfer* message to the RCA which results in state $U9$. The *SymKey* in this message may or may not be useful to decrypt the content previously delivered but that detail is immaterial because the content is not what the downloader expected to get. When the downloader sends the *ContentRcvd* message to the RCA with the digest of the received content, unsafe state $U10$ results. At this point, the RCA checks the message digest of the actual content with the one in *ContentRcvd* and concludes that incorrect content has been sent to the downloader. The effect is that the downloader times out waiting for the *KeyDelivery* message from the RCA and the protocol returns to the initial state $S0$, defeating the attacks.

In state $S5$ after *ContentDelivery* by a server, a selfish peer can launch attacks $a3$ and $a4$ to claim the upload credit (UC) instead of the server (perhaps because it also has the content denoted by *content_id*). It can do so by forging the *KeyTransfer* message or replaying an old one and causing the protocol to enter unsafe state $U11$. If the *Key-*

Transfer message was just encrypted, the RCA could not distinguish a valid one from a forged or replayed message. However, since the *KeyTransfer* message includes a signature that identifies the server, downloader, time stamp, and the content_id, these attacks will fail and the protocol will resume in state $S5$. There is a possibility that a valid *KeyTransfer* message will arrive at the RCA in state $U11$, causing the system state to change to $U12$. Since the RCA can distinguish a valid message from a forged or replayed one, the net effect is for the protocol to resume in safe state $S6$.

To avoid debit for the received content, a selfish downloader can launch attack $a5$ in state $S6$. It can do so by forging its identity in the *ContentRcvd* message, causing the protocol to enter unsafe state $U13$. Since in this case, the identity of the downloader in the *ContentRcvd* message will fail to match that in *KeyTransfer* message, this attack will fail and the protocol will revert back to safe initial state $S0$.

Since *KeyDelivery* is encrypted by the RCA, no attack can be launched in $S7$. It is also important to note that snooping of *ContentDelivery*, *KeyTransfer*, and *KeyDelivery* is possible by selfish peers in order to launch attack $a5$ but can cause no harm to the protocol because all three are encrypted. As a result, no state changes are required if these messages are snooped.

6.2.3. Interaction r-sr

Periodically, the RCA starts in state $S0$ and processes the upload credit (UC) and the download debit (DD) for the servers and the downloaders using the maintained *transaction state*. Upon finishing the processing, it sends the UC to the respective servers, causing the protocol to enter state $S8$. When the servers acknowledge the receipt of UC, state $S0$ is restored.

Attacks $a0$ and $a3$ can be launched by selfish peers by snooping on the UCs in transit or by modification. Since the UC is signed, snooping does not cause damage to the protocol operation. However, modification causes the protocol to enter unsafe state $U14$. When the UC does not reach the server, the RCA times out waiting for the acknowledgment of the UC. The result is that the initial protocol state $S0$ is restored and the RCA can resend the UC. In any event, both the attacks are defeated.

The specification of strong reputations satisfies the *convergence* and *protection* conditions. The convergence condition is satisfied because every unsafe state Ui, $i \in \mathcal{I}$ resolves to Si, $i \in \mathcal{I}$. The protection condition is satisfied because the critical variables: transaction state maintained at the RCA, searcher_id, server_id, downloader_id, content_id, and time_stamp are not modified in any unsafe state.

7. Weak Reputations

Reputation tracking in weak reputations is also done in two steps, however the details differ from those for strong reputations (section 5): 1) peers enrolled in reputation tracking now save proofs of their contributions both during P2P search and download functionalities to collect query-response credit (QRC) and upload credit (UC) and 2) they periodically send these proofs to the RCA. The RCA processes UC, the corresponding download debit (DD) and the QRC and sends the encrypted reputations to relevant peers for keeping locally. Just as in the case of strong reputations, the RCA stores the DD with itself to ensure that peers do not drop them. We now describe the proof of contributions saved by the peers and the processing at the RCA.

7.1. Proof of Peer Contributions

pSearch: For every query-response message processed during the content search, a searcher peer saves $\{$*searcher_identity*, *query_keywords*, *time_stamp*, *querier_identity*$\}_{SK_{query}}$ as the *proof of searching (pSearch)*, just as in the case of strong reputations.

pServe: For saving the *proof of serving (pServe)*, the following exchange takes place between the enrolled downloader and the enrolled server peer at the time of the file download[12]:

- The downloader sends a *requester por-*

[12] It is important to note that if *any* of the peers are not enrolled, the following exchange does not take place. This means that by creating additional *unenrolled* identities peers cannot earn any credit to their reputations.

tion of the receipt (RPR) in the form of $\{$*downloader_identity*, *server_identity*, *file_name*, *file_size*, *time_stamp*$\}_{SK_{download}}$ to the sender peer.

- The server peer verifies the information using PK_s and stores $\{$*downloader_identity*, *server_identity*, *file_name*, *file_size*, *time_stamp*$\}_{SK_{download}}\}_{SK_{serve}}$ as pServe. At that point, it serves the content to the downloader.

7.2. Processing at the RCA

Periodically, the enrolled peers send the pSearchs and pServes collected thus far to the RCA. The RCA uses these to maintain *transaction state* of the form (*downloader_identity*, *server_identity*, *file_name*, *file_size*, *time_stamp*, *credit_processed_list*). The credit_processed_list is similar to that in strong reputations. Notice that since the RCA does not interfere with the search and download functionality, as it does in the case of strong reputations, it can not maintain any state by itself. It relies on the pServes to maintain the transaction state about downloads.

Processing upload credit (UC) and download debit (DD): The RCA in weak reputations uses pServes to infer the UC for the serving peers and the corresponding DD for the downloading peers. Just as for strong reputations, it then updates the credit_processed_list and sends signed reputations of the form $\{$*RCA_identity*, *time_stamp*, *reputation_score*, *server_identity*$\}_{SK_{RCA}}$ to the relevant serving peers. To avoid having the downloaders drop negative reputation scores, the RCA retains the DDs in the form of *debit state* with itself until those peers send some credits for processing, just like in strong reputations.

Processing query-response credit (QRC): Since peers can forge QRCs for files that were never downloaded, the QRC from the pSearchs is not processed until the corresponding UCs and DDs are processed, even if they arrive before the pServes. After inferring QRCs using pSearchs and the transaction state, the RCA updates the credit_processed_list and sends an encrypted reputation score of the form $\{$*RCA_identity*, *time_stamp*, *reputation_score*, *searcher_identity*$\}_{SK_{RCA}}$ to the searchers.

8. Attack Analysis for Weak Reputations

Just like in the case of strong reputations, the security mechanisms in weak reputations are mainly designed to counter attacks launched by enrolled peers that are selfish, not malicious. The discussion in section 10 about enrolled peers that are not malicious or selfish but agree to suffering a penalty for others' sake applies equally well to weak reputations.

The signing of pSearchs for QRC and pServes for UC provides protection against message forgery and modification attacks. The *credit_processed_list* maintained by the RCA along with the time stamps in the pSearchs and pServes guard against message replay attacks. These ensure that peers can only collect credit once and that only peers that send the signed pServes and pSearchs can collect the credit. As a result, the last property out of the 4 properties listed in section 6 is satisfied. However, the rest of the 3 properties are not satisfied for weak reputations because of the possibility of the following attacks:

Collusion to collect QRC: Peers collect pSearchs while processing content search queries without the intervention of any overseeing authority. The RCA postpones processing of pSearchs sent by the peers until it receives the corresponding pServes. This precludes peers from generating pSearchs for downloads that did not take place in the system. However, peers can collude with each other to collect pSearchs (and hence QRC) by supplying information about the queries that any of them have processed since the RCA does not have a way of verifying the pSearchs. In practice such a collusion is not worth it for two reasons: 1) the main issue the reputations help address is that of motiving peers to contribute to the system. Peers have ample opportunity to cooperate in the system and earn QRC for processing queries they encounter and 2) since the QRC is expected to be very small as compared to the UC (as described in section 2)

such a collusion would not be very beneficial.

Limited content reliability: In order to allow the server peers to collect credit for serving content, the downloaders are required to send the receiver portion of the receipt (RPR) as described in section 7.1 before they can receive the content from the server. This ensures that the downloaders cannot avoid a debit to their reputations if they consume servers' resources. However, the exchange for collecting pServe does not protect the downloaders from avoiding debits in cases where they do not get the requested content in its entirety either due to servers' selfishness or network failures. As as result, there is limited content reliability for the downloaders in weak reputations. The strong reputations provide content reliability but at a significant cost. To circumvent the problem in a light-weight manner, the weak reputations allow the downloaders to report such servers' to the RCA. Upon receiving many such complaints, such peers may be black-listed. Though the system offers no advantages for doing so, the misreporting is prone to peers colluding to malign the reputation of certain targeted peers and is a hard problem to solve in general.

Though the exchange to collect pServe offers only limited content reliability, including the identity of the servers in the RPR guarantees that the server peers do not collude with other peers to share the RPR. Otherwise, multiple peers could collect pServes and each could cause a debit for the downloader. The downloader can choose to generate RPRs even for the transfers that did not take place but doing so would only cause more debits to be accumulated in its account with the RCA.

9. Overheads

9.1. Upper Bounds on Overheads in Strong Reputations

In comparison with the Gnutella protocol, the strong reputations incur five types of overheads each time content is downloaded: 1) extra messages during content download, 2) messages exchanged periodically between the RCA, searchers, and servers for reputation computation purposes, 3) extra computations in order to encrypt and decrypt messages using public and shared key cryptography and in performing hash computations, 4) crawl overhead for the RCA in order to obtain periodic snapshots of P2P network topology, and 5) storage overhead at the RCA to store copies of files sent by content creators.

Figure 2 summarizes the upper bounds on each of these overheads for *each download* assuming that all peers in a P2P network are enrolled in reputation computations. The RCA crawl overheads are not considered because they depend on the number and placement of RCAs and the crawl frequency. We also ignore storage overheads at the RCA. In figure 2, the average number of neighbors each peer is connected to is denoted by n, and the *hop-count* for queries is denoted by h. We now describe the entries of the table.

Extra messages: Strong reputations do not require any extra messages during content search. Content download in Gnutella requires a request to download content before the actual download. In addition to this, the strong reputations protocol requires three messages during content download: 1) *KeyTransfer*, 2) *ContentRcvd*, and 3) *KeyDelivery*.

Using the transaction state, the RCA infers UC and DD periodically. It then sends a message containing the UC to each server. Including the acknowledgment for this message, two such messages are required for each content download. This is an upper bound because if some servers have contributed to multiple downloads in a period their UCs can be aggregated, bringing the overall messages required in strong reputations.

Additionally, up to $\sum_{k=1}^{h} n^k$ searchers can send pSearchs for collecting QRC for each download. In practice, searchers collect pSearchs and send them together periodically, so the average number of messages will be lesser. Sending the corresponding QRCs and receiving their acknowledgments by the RCA requires $2\sum_{k=1}^{h} n^k$ messages.

The first row of table in figure 2 presents the extra messages during search, download, and reputation computations. The upper bound on the total number of extra messages required for each successful content download in strong reputations is: $3+2+\sum_{k=1}^{h} n^k + 2\sum_{k=1}^{h} n^k = 5+3\sum_{k=1}^{h} n^k$.

Security related operations: We next turn

Table 2
Upper bounds on overheads *per download* in strong reputations.

	Search	Download	Reputation related
Extra messages	0	3	$2 + 3\sum_{k=1}^{h} n^k$
Public key encryptions	$\sum_{k=1}^{h} n^k$	5	$1 + \sum_{k=1}^{h} n^k$
Public key decryptions	0	4	$1 + 2\sum_{k=1}^{h} n^k$
Symmetric key operations	0	3	0
Hash computations	0	2	0

our attention to quantifying the security related operations required in strong reputations for each download. In contrast, Gnutella does not require any such operations. In symmetric key cryptography, encryption and decryption are symmetric. However, the encryption operations in public key cryptography are about 100 times slower than the decryption. Digest computations are typically $10,000$ times faster than the public key encryption. Due to the difference in processing for each of these operations we list these overheads separately in figure 2.

In generating the pSearchs each searcher performs one signing operation using its private key, leading to up to $\sum_{k=1}^{h} n^k$ public key encryption operations during content search. Content download in strong reputations involves both public and symmetric key operations and also digest computations. Steps 1 through 5 during content download require 9 public key operations (5 encryptions and 4 decryptions), 3 symmetric key operations, and 2 hash computations.

The RCA decrypts the pSearchs before generating QRCs and signs the QRCs before sending them. Upon receipts of the QRCs, the searchers decrypt the QRCs. Thus, the upper bound on the number of public key encryptions and decryptions during content search are $\sum_{k=1}^{h} n^k$ and $2\sum_{k=1}^{h} n^k$ respectively. Similarly, the RCA encrypts the UC before sending it to the server and the server decrypts it, leading to 1 encryption and 1 decryption per download.

9.2. Upper Bounds on Overheads in Weak Reputations

Compared to the strong reputations the weak reputations have fewer overheads per download. In particular, they do not have RCA crawl overhead or the overhead for symmetric key operations and hash computations.

Figure 3 summarizes the upper bounds on the extra messages and public key operations for *each download* assuming that all peers in a P2P network are enrolled in reputation computations. Just as in the case of strong reputations, the average number of neighbors each peer is connected to is denoted by n and the *hop-count* for queries is denoted by h. We now describe the entries of the table.

Extra messages: Weak reputations also do not have any extra messages during content search. However, during content download they require one extra message to be sent by the downloader.

The RCA infers UC and DD periodically using the pServes sent by the servers. It then sends a message containing the UC to each server. Including the acknowledgment for this message, three extra messages are required for reputation inference for each content download. Additionally, up to $\sum_{k=1}^{h} n^k$ searchers can send pSearchs for collecting QRC for each download. Sending the corresponding QRCs and receiving their acknowledgments by the RCA requires $2\sum_{k=1}^{h} n^k$ additional messages.

The first row of the table in the figure 3 presents the extra messages during search, download, and reputation computations. The upper bound on the total number of extra messages required for each successful content download in weak reputations is: $1 + 3 + 3\sum_{k=1}^{h} n^k = 4 + 3\sum_{k=1}^{h} n^k$. Compared to the strong reputations, there is one less message *per* content download.

Security related operations: In generating the pSearchs each searcher performs one signing

Table 3
Upper bounds on overheads *per download* in weak reputations.

	Search	Download	Reputation related
Extra messages	0	1	$3 + 3\sum_{k=1}^{h} n^k$
Public key encryptions	$\sum_{k=1}^{h} n^k$	2	$1 + \sum_{k=1}^{h} n^k$
Public key decryptions	0	1	$2 + 2\sum_{k=1}^{h} n^k$
Symmetric key operations	0	0	0
Hash computations	0	0	0

operation using its private key, leading to up to $\sum_{k=1}^{h} n^k$ public key encryption operations during content search. Content download in weak reputations involves 2 public key encryptions and 1 public key decryption.

Further, The RCA decrypts the pSearchs before generating QRCs and signs the QRCs before sending them. Upon receipt of the QRCs, the searchers decrypts the QRCs. Thus, the upper bound on the number of public key encryptions and decryptions during content search are $\sum_{k=1}^{h} n^k$ and $2\sum_{k=1}^{h} n^k$ respectively. Further, 1 encryption and decryption each is required for sending and receiving the UC. Compared to strong reputations the weak reputations incur lesser computational overheads for security related operations.

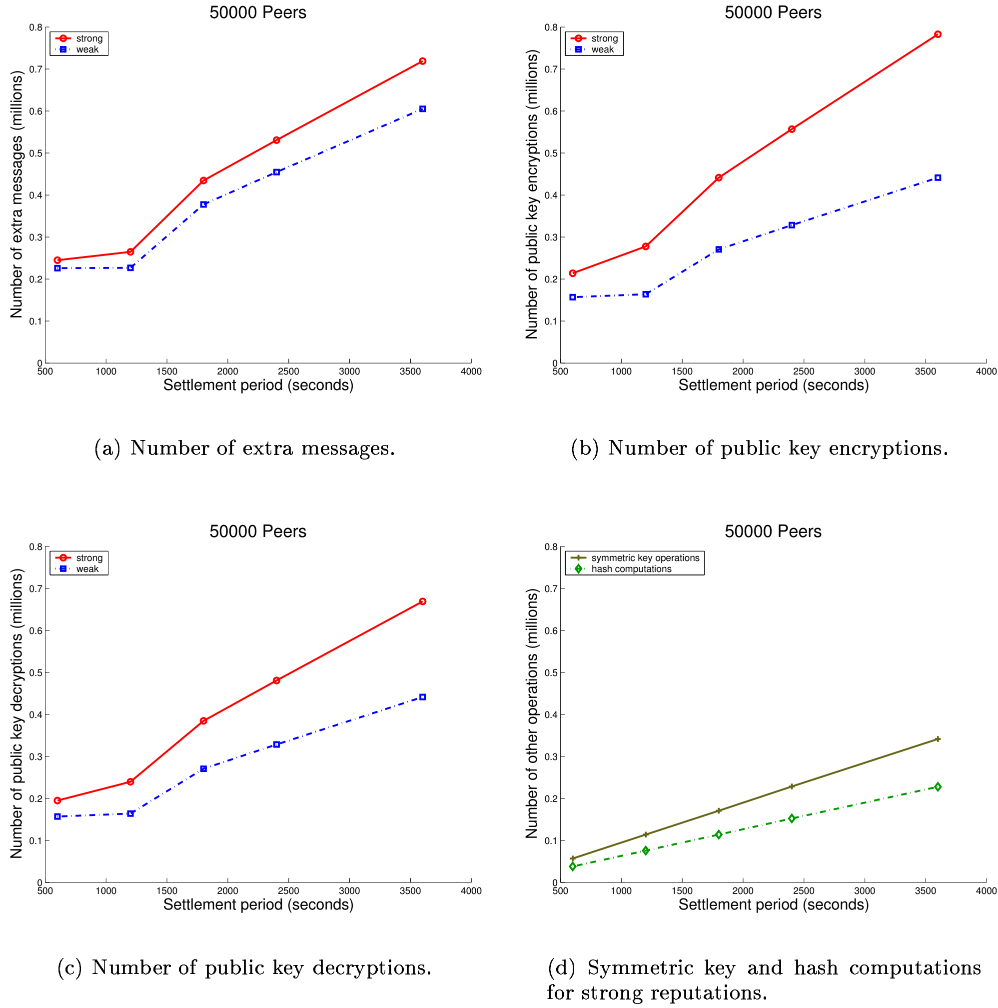
9.3. Comparison of Overheads Through Simulations

To compare the overheads between strong and weak reputations through simulations, we generated connected topologies with peer populations ranging from 5000 to $50,000$. Each peer in these topologies is connected on an average to about 4 other peers in the system. We assume that the total number of files in each topology is the same as the peer population. The file popularities are Zipf distributed with a parameter of 1.0 and the file sizes are uniformly distributed between 0-8MBytes.

The requests for files by peers are uniformly distributed among all the peers with an exponential inter-arrival time of 50 requests/second. To create multiple logical small-worlds where some peers serve more files than the others we created 100 groups of peers for each population size. Within each group the number of cached files are Zipf distributed with a parameter of 1.0. While creating connections in each topology, we randomly picked peers and did not attempt to distribute peers with certain numbers of files to share in any particular way. The simulation logs are 1 hour long for each topology. The number of hops queries are allowed to go varied from 4 hops to 7 hops. The number of hops were constant for all peers for one run.

For the evaluations, we experimented with population sizes of 5000, $10,000$, $25,000$ and $50,000$. With the number of hops staying constant, the effect of varying population sizes was that the smaller the peer population in the topology, the more the successes during content search (and hence lesser failures). This is expected because for smaller topologies, the number of copies of files in the system are more. Since our goal is to compare the strong and weak reputations for their overheads, as long as they are both tested under same conditions the conclusions are not expected to change.

We now present the results of the topology with $50,000$ peers where each query is allowed to go 7 hops. The *settlement period* (the periodicity at which the RCA processes the reputation related debits and credits) varies between 600 seconds to 3600 seconds for the graphs shown in figures 5(a), 5(b), 5(c), and 5(d). Figure 5(a) shows the difference in the number of total extra messages in the system between strong and weak reputations.



(a) Number of extra messages.

(b) Number of public key encryptions.

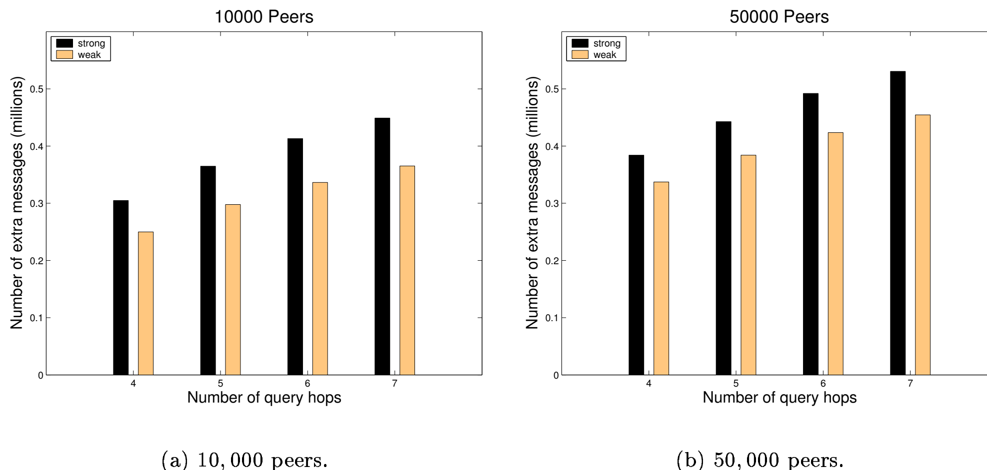
(c) Number of public key decryptions.

(d) Symmetric key and hash computations for strong reputations.

Figure 5. Overheads of strong and weak reputations.

Figure 5(b) and 5(c) show the difference in the total number of encryption and decryption operations between the two schemes respectively. Figure 5(d) shows the total number of symmetric key operations and hash computations in the system for strong reputations. Weak reputations do not have this overhead. The RCA crawl overheads required in strong reputations were not analyzed.

In all the graphs the total overheads as well as the difference in overheads between strong and weak reputations increases with the increase in settlement period. This is because the number of total downloads increase with the settlement period, leading to higher overall overheads on the system. Further, the downloads contribute most to the difference in overheads between strong and weak reputations, which leads to more divergence between strong and weak reputations.



(a) 10,000 peers.

(b) 50,000 peers.

Figure 6. Effect of varying the number of hops.

Figures 6(a) and 6(b) show the affect of varying hops on the overheads for strong and weak reputations for topologies with 10,000 and 50,000 peers respectively. The settlement period was set to 2400 seconds for these graphs. As expected, with increase in the number of hops the queries are allowed to go, the overheads increase. Also, the difference in overheads between strong and weak reputations stayed constant.

10. Deployment Considerations

We now discus the practical considerations in incorporating strong and weak reputation tracking schemes in deployed instances of P2P networks like Gnutella and Kazaa.

Multiple identities: Since the enrollment in reputation tracking is voluntary, the schemes cannot guard against multiple identities. Even enrolled peers who cannot create more than one enrolled identity can create other identities that are not enrolled in the reputation tracking. However, since the RCA processes QRC, UC, and SC only for the enrolled peers, no credit can be earned in the system by the unenrolled identities.

Collusion: In analyzing the attacks possible on the proposed reputation tracking schemes, we focussed only on the selfish peers. Since the underlying Gnutella infrastructure itself is vulnerable to attacks from the malicious peers, we chose not to focus on them for reputation tracking purposes. However, there is a third category of peers that deserves attention. Certain enrolled peers may be willing to incur debits to their reputation scores in order to help other targeted peers through collusion. They can do so by agreeing on fake content downloads. Though the RCA can detect simple collusion of this nature, in general it cannot prevent a collusion where certain enrolled peers agree to penalizing themselves for the sake of others. However, given that most peers are expected to be *unknown* to each other, an occurrence of such a collusion is expected to be insignificant in practice.

Inaccuracies in reputation computations: There are several sources of inaccuracies in reputation computations. Since the RCA does not synchronize the processing of pSearchs, by the time certain peers choose to contact the RCA to obtain QRCs, the corresponding transaction state may have been erased due to the upper bounds

on memory requirements for storing it. Similarly, the download debit state maintained by the RCA may be lost by the time those peers contact the RCA with credit requests. In fact, peers can avoid debits stored at the RCA by not contacting the RCA with credit requests for a long enough duration such that the debit state gets erased (although, they will also not be able to collect any credit for their work in the system in the meanwhile).

A P2P transaction between an enrolled peer and an unenrolled peer causes the enrolled peer to be not be able to collect the credit for its work in the system. Thus, the enrollment of peers in the reputation computations also impacts the accuracy of reputation computations. The net result of all these sources of inaccuracies is the decreased accuracy of reputation computations. It is important for the applications using peer reputations as a substrate to understand these trade-offs.

Reliance on the RCA infrastructure: The proposed schemes rely on the RCA infrastructure for reliable reputation tracking. The deployed Gnutella-like P2P networks depend on an infrastructure of *voluntary* bootstrapping servers [18] to allow new peers to join the system. The existing bootstrapping infrastructure can be enhanced to provide the RCA functionality. However, since the RCA needs to be trustworthy one cannot depend on voluntary compliance.

Choosing an RCA: In describing the schemes we assumed that the RCA is a centralized entity. Clearly, this assumption is not robust against failures. One way to share the load among an infrastructure of RCAs is for each RCA to be made responsible for certain number or types of files. This would distribute the load among the RCAs. The RCAs dealing with a certain number or types of files can further be replicated for robustness. However, doing so would require synchronization of transaction state among them.

Consolidation of reputations: Periodically, when the peers send the pSearchs to the RCA, they collect encrypted and time-stamped reputation scores to be stored locally. For ease of presenting aggregate reputation scores that the applications using reputation scores may require, the RCA can perform consolidation of reputation scores.

Though reputations can be saved across sessions, to prevent peers from earning a good reputation once and never contributing resources again to the P2P system, upper bounds on the life of reputations need to be defined. These upper bounds can also be used during consolidation of reputations.

Reputations for new peers: All peers start with a reputation score of 0. The new peers start earning reputation by placing content in their shared directory or by serving the downloaded content. When their uploads exceed the downloads, their reputation will become non-negative. Further, they can earn QRC for simply being a part of the system.

11. Conclusion

While many proposals to track peer reputations in P2P networks are available, to our knowledge this is the first work that explores the trade-offs in reliability and overheads in reputation tracking. In the context of reputation tracking using objective criteria (as described in [9]) and an infrastructure of RCAs, this paper proposes two methods of reputation tracking that vary in overheads and reliability.

Though the *strong reputations* proposed in this paper track reputations with formally verifiable reliability, they introduce a centralized bottleneck during content download, require the RCA infrastructure to crawl the P2P network to maintain snapshots of the topology, and incur higher security related computational overheads on the underlying P2P network compared to the *weak reputations.* The RCA infrastructure in weak reputations on the other hand is not a bottleneck for search and download functionality and only needs to be contacted periodically. While strong reputations may be feasible for small P2P networks, their overheads could compromise the scalability of larger P2P networks. The overheads and reliability trade-offs in weak reputations present a more compelling alternative for potential deployment.

REFERENCES

1. K. Lai, M. Feldman, I. Stoica, J. Chuang, Incentives for cooperation in peer-to-peer networks, in: Workshop on Economics of Peer-to-Peer Systems, 2003.
2. M. Gupta, M. H. Ammar, Service differentiation in peer-to-peer networks utilizing reputations, in: ACM NGC, 2003.
3. K. Aberer, Z. Despotovic, Managing trust in a peer-2-peer information system, in: Ninth International Conference on Information and Knowledge Management (CIKM), 2001.
4. E. Damiani, S. D. C. di Vimercati, S. Paraboschi, P. Samarati, F. Violante, A reputation-based approach for choosing reliable resources in peer-to-peer networks, in: 9th ACM Conference on Computer and Communications Security, 2002.
5. S. D. Kamvar, M. Schlosser, H. Garcia-Molina, EigenRep: Reputation management in P2P networks, in: World-wide Web Conference, 2003.
6. S. Lee, R. Sherwood, B. Bhattacharjee, Cooperative peer groups in NICE, in: IEEE INFOCOM, 2003.
7. L. Xiong, L. Liu, Building trust in decentralized peer-to-peer communities, in: International Conference on Electronic Commerce Research (ICECR-5), 2002.
8. Kazaa participation level, `http://www.kazaa.com/`.
9. M. Gupta, P. Judge, M. H. Ammar, A reputation system for peer-to-peer networks, in: ACM NOSSDAV, 2003.
10. V. Vishnumurthy, S. Chandrakumar, E. G. Sirer, KARMA : A secure economic framework for peer-to-peer resource sharing, in: Workshop on Economics of Peer-to-Peer Systems, 2003.
11. R. Morselli, J. Katz, B. Bhattacharjee, A game-theoretic framework for analyzing trust-inference protocols, in: Workshop on Economics of Peer-to-Peer Systems, 2004.
12. M. Ripeanu, I. Foster, A. Iamnitchi, Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design, IEEE Internet Computing Journal 6 (1).
URL `citeseer.nj.nec.com/ripeanu02mapping.html`
13. P. McDaniel, S. Jamin, A scalable key distribution hierarchy, Tech. Rep. CSE-TR-366-98, Electrical Engineering and Computer Science, University of Michigan (1998).
14. R. Rivest, The md5 message digest algorithm, RFC 1321 (Apr. 1992).
15. P. J. D. Eastlake, Us secure hash algorithm 1, RFC 3174 (Sep. 2001).
16. M. G. Gouda, Elements of network protocol design, John Wiley & Sons, 1998.
17. D. F. Lee, A formal presentation of electronic commerce protocols, Ph.D. thesis, The university of Texas at Austin (2002).
18. Gnutella web caching system, `http://www.gnucleus.com/gwebcache/`.

Appendix: Formal Specification for Strong Reputations

```
process querier[i: 0...n-1]
inp  K^-1_self, K_self : integer;          {own keys}
     K_RCA             : integer;          {RCA's public key}
     K                 : array [0...n-1] of integer; {public keys of all peers}
var  new_query         : boolean;
     keywords          : string;
     reply             : integer;
par  j                 : array [0...n-1] of integer;     {for peers}
begin
     new_query = true →                    {new query generated}
          keywords := any;
          send query(keywords) to searcher[j]; {send to all direct neighbors}
          new_query := false;
     | timeout reply = 0 →                 {ready to resend query}
          new_query := true;
end
```

```
process searcher[i: 0...n-1]
inp    K^-1_self, K_self : integer;       {own keys}
       K_RCA             : integer;       {RCA's public key}
       K                 : array [0...n-1] of integer; {public keys of all peers}
var    timer                                  : boolean;
       pSearch                                : array [0...p-1] of string; {#pSearchs< p}
       keywords', time_stamp, credit', ack    : string;
       querierID                              : integer;
       reputation                             : array [0...p-1] of real; {#reputations < p}
par    j                 : array [0...n-1] of integer;          {for peers}
begin  rcv query(keywords') from querier[j] →
            pSearch[k] := NCR(K^-1,(MD5(K_self | querierID | keywords' | time_stamp)));
                                   {save pSearch}
       | timeout timer = true →
            send pSearch[k] to RCA;
            timer := false;
       | rcv credit' from RCA →
            reputation[k] := DCR(K_RCA, credit');          {save reputation}
            send ack to RCA;
end
```

```
process downloader[i: 0...n-1]
inp  K^-1_self, K_self : integer;        {own keys}
     K_RCA             : integer;        {RCA's public key}
     K                 : array [0...n-1] of integer {public keys of all peers}
var  select, timer                            : boolean;
     content'                                 : binary;
     time_stamp, msg'                         : string;
     contentID, RCA_ID, serverID, symKey      : integer;
par  j                 : array [0...n-1] of integer;        {for peers}
begin
     select = true →                     {a server selected for download}
          send downloadReq(NCR(K^-1_self, (MD5(K_self) | contentID
                                         | time_stamp))) to server[j];
          select := false;
     | rcv contentDelivery(content') from server[j] →
          send contentRcvd(NCR(K^-1_self, (serverID | MD5(K_self) | contentID
                                 | MD5(content') | time_stamp))) to RCA;
     | timeout timer = true →
          select := true;                               {resend request}
          timer := false;
     | rcv keyDelivery(msg') from RCA →
          (contentID, RCA_ID, symKey) := DCR(K^-1_self, (msg'));
          DCR(symKey, content);                         {decrypt content}
end
```

```
process server[i: 0...n-1]
inp  K^-1_self, K_self : integer;        {own keys}
     K_RIA             : integer;        {RIA's public key}
     K                 : array [0...n-1] of integer; {public keys of all peers}
var  content                                   : array [0...m-1] of binary;
     time_stamp, request', ack                 : string;
     contentID, downloaderID, symKey, credit'  : integer;
     reputation                                : array [0...p-1] of real;{#reputations < p}
par  j                 : array [0...n-1] of integer;        {for peers}
begin
     rcv downloadReq(request') from downloader[j] →
          (downloaderID, contentID, time_stamp) := DCR(K_j, (request'));
          symKey := DES(contentID | downloaderID | time_stamp);
                    {generate symKey using DES}
          send contentDelivery(NCR(symKey, (content[contentID]))) to downloader[j];
          send keyTransfer(NCR(K_RIA, NCR(K^-1self, (MD5(K_self)|downloaderID
                         |contentID|time_stamp))|symKey)) to RIA;
     | rcv credit' from RIA →
          reputation[k] := DCR(K_RIA, (credit'));          save reputation
          send ack to RCA;
end
```

```
process RCA
inp  K^{-1}_{RCA}, K_{RCA}: integer;        {own keys}
     K                  : array [0...n-1] of integer {public keys of all peers}
var  content                                   : array [0...m-1] of binary;
     keyRcvd, digestRcvd                       : array [0...n-1] of boolean;
     time_stamp, trans_his                     : string
     symKey, contentID, serverID               : array [0...n-1] of integer;
     digest, msg'                              : integer;
     downloaderID, searcherID                  : integer;
     reputation, credit, debit                 : array [0...j-1] of real;
par  j, l               : array [0...n-1] of integer;          {for peers}
begin rcv keyTransfer(msg') from server[j] →
        (serverID[j], downloaderID, contentID[j], symKey[j] := DCR(K^{-1}_{RCA}, (msg'));
        if digestRcvd[j] := true →{content digest already rcvd}
            send keyDelivery(NCR(K_j, (contentID[j] | RCA_ID | symKey[j]))) to downloader[j];
            digestRcvd[j] := false;
            update(trans_his);                        {update transaction history}
        fi
     | rcv contentReceived(msg') from downloader[j] →
        (serverID[j], downloaderID, contentID[j], digest, time_stamp):= DCR(K_j,(msg'));
        if keyRcvd[j] := true & digest = MD5(NCR(symKey[j], (content[contentID[j]]))) →
            send keyDelivery(NCR(K_j, (contentID[j] | RCA_ID | symKey[j]))) to downloader[j];
            update(trans_his);
            keyRcvd[j] := false;
        fi
     | timeout timer = true →      {time to process debits/credits}
        process(trans_his);        {timer for bill acks not shown}
        reputation[j] := NCR(K^{-1}_{RCA},(serverID[j] | contentID[j] | time_stamp | credit[j])) {one msg/peer}
        if reputation[j]-debit[j] >= 0 →
            send reputation[j]-debit[j] to to server[j];
            debit[l];              {save corresponding debit for the downloader[l]}
        fi
        timer := false;
     | rcv pSearch(msg') from searcher[j] →
        if match(trans_his, DCR(K_j, (msg'))) = false → skip {state erased}
        | match(trans_his, DCR(K_j, (msg'))) = true →
            send (NCR(K^{-1}_{RCA},(searcherID | contentID[j] | time_stamp | credit[j]))) to searcher[j]
        fi
end
```

In *Abstract Protocol Notation*, each entity of the protocol is denoted by a *process*. Each process p in turn is defined by a set of *constants*, *inputs*, *variables*, *parameters*, and *actions*. The constants of p have fixed values while the inputs can be read but not updated. The variables can be read and updated by the actions of p. A parameter declared in a process is used to write a finite set of actions as one action, with one action for each possible value of the parameter. Each *action* of a process is of the form $< guard > \rightarrow < statement >$. The *guard* of an action is one of the 3 forms: 1) a boolean expression over the constants and variables of p, 2) a receive guard of the form ***rcv***$< message >$***from*** q; where q is another process in the protocol, and 3) a *timeout* that contains a boolean expression over the constants and variables of every process and the contents of all channels in a protocol. An action is executed only when its guard is true and consists of executing the *statement* of this action. Details on the types of variables and statements can be found in [16]. Comments can be added anywhere in a process definition; each comment is placed between the two brackets { and }.