Implementing a Reputation-Aware Gnutella Servent

Fabrizio Cornelli¹, Ernesto Damiani¹, Sabrina De Capitani di Vimercati², Stefano Paraboschi³, and Pierangelo Samarati¹

¹ Dip. Tecnologie dell’Informazione, Università di Milano, 2013 Crema - Italy
fcornelli@crema.unimi.it, {damiani,samarati}@dti.unimi.it
² Dip. Elettronica per l’Automazione, Università di Brescia, 25123 Brescia - Italy
decapita@ing.unibs.it
³ Dipartimento di Elettronica e Informazione
Politecnico di Milano
20133 Milano - Italy
parabosc@elet.polimi.it

Abstract. We describe the design and implementation of a reputation-aware servent for Gnutella-like peer-to-peer systems.

1 Introduction

Peer-to-peer (P2P) architectures have recently become the subject of considerable interest, both in the population of Internet users and in the research community [6,8,12,11,15,16,21]. Internet users find in P2P applications a convenient solution for the anonymous exchange of resources. The research community has looked with interest to the huge success that these applications were achieving and it has started to investigate many issues that arise in this context, like performance, usability, and robustness. These architectures appear as an interesting paradigm for the development of many novel network applications.

One of the urgent issues to be investigated is the definition of security mechanisms that would permit to reduce the risks that are currently faced by users of these applications. Anonymity of the interaction is one of the major reasons of the success of these solutions, but it usually implies that no guarantee can be assumed on the quality of resources available on the network. In previous work we have designed a protocol [5] that focused on this issue and offered a mechanism that, even in the presence of anonymous participants, permits to ask the user community an opinion on a particular node. The reputation [15] of the node can then be the basis on which a user can assess the risk of using a resource retrieved by the network. The protocol is designed following traditional guidelines for secure protocols and is robust against attacks by malicious users, who can badmouth reputable nodes or try to build a good reputation for a tampered with resource.

In [5] we presented the main features of the protocol. In this paper we discuss the implementation of the protocol and its integration with a P2P application.
2 P2PRep protocol

In a traditional Gnutella interchange, a servant $p$ looking for a resource broadcasts to all its neighbors a Query message reporting the search keywords. Each servant receiving the Query and with resources matching the request, responds with a QueryHit message. The QueryHit message includes the number of files num_hits that matched the keywords, a set of triples Result containing the files’ names and related information, the speed in Kb/second of the responder, the servant_id of the responder, a trailer with application-specific data, and the pair (IP_port) to be used to download the files. Based on the offers’ quality (e.g., number of hits and declared connection speed), as well as on possible preference criteria, $p$ then chooses the servant from which to execute the download and directly contacts it for the download.

Our proposal, called P2PRep, enhances this process by providing a reputation-based protocol by which $p$ can assess the reliability of an offerer before actually downloading a resource from it. The basic idea is very simple. After a servant downloads a resource from another servant, it can record whether or not the download has completed to its satisfaction. Before downloading a resource from another servant, a servant can poll its peers about their knowledge of the offerer, thus assessing its reputation. If that offerer is then chosen, the downloading servant can update its local recording of both the offerer (recording whether the downloaded resource was satisfactory or not) and the peers that expressed an opinion on it (recording whether their opinion on the offerer matched the final outcome – credibility).

The realization of the protocol requires some considerations. First, in order to keep track of a servant’s reputation and credibility, identifiers’ persistency must be assumed. In traditional Gnutella, the identifier with which a servant presents itself can, in principle, change at every session. However, a servant wishing to establish some reputation as offerer or voter is encouraged to maintain its identifier persistent. Note that this does not imply that the servant’s identity is disclosed, as the declared identifier works only as a pseudonym (opaque identifier).\(^4\) Second, care must be taken to ensure confidentiality and integrity of the messages being exchanged. To this purpose, our protocol uses public key encryption. In particular, a servant’s identifier is assumed to be the digest of a public key, obtained using a secure hash function, and for which the servant knows the corresponding private key. Exchanged messages are assumed to be signed with a secret key established by the sender (when integrity must be ensured) and encrypted with a public key established by the recipient (when message confidentiality must be ensured). The key ring used to these purposes can be the pair associated with a servant’s identifier or can be generated ad-hoc for the exchange (if the identifier of the involved servant is not to be disclosed).

\(^4\) The ability of changing pseudonyms at any time, makes it possible for malicious peers to not be recognized from one interaction to another simply by changing their identifier. This is not a problem as by changing their identifiers, such peers will start as new, with no reputation at all, and they are therefore unlikely to be chosen.
Protocol 1  P2PRep protocol

Initiator: Servent p

Peers: Participants in the message broadcasting, among which a set O of offerers and a set V of voters

INITIATOR

Phase 1: Resource searching

(G)  1.1 Start a search request by broadcasting a query message
    query(search_string, min_speed)

(G)  1.2 Receive a set of offers from offerers O
    queryOffer(num_hits, IP, port, speed, Result, trailer, servantId)

Phase 2: Polling

2.1 Select top list T \( \subseteq \) O of offerers

2.2 Generate a pair of public, secret keys (PKoff, SKoff)

2.3 Poll peers about the reputations of offerers T
    poll(T, PKoff)

2.4 Receive a set of votes from voters V
    pollReply([(IP, port, Votes, servantId)])[SKi, PKi]PKoff

Phase 3: Vote evaluation

3.1 Remove from V voters that appear suspicious (e.g., checking IP addresses)

3.2 Select a random set V' \( \subseteq \) V of voters and check their identity by sending message
    areYou(response)

3.3 Expect back confirmation messages from each selected voter

3.4 Send a challenge message to s
    challenge(r)

3.5 Receive a response message from s containing its public key PKs and the challenge
    signed with its private key SKs
    response[r, SKs, PKs]

3.6 If the challenge-response exchange fails, terminate the process

Phase 4: Resource downloading

4.1 Select servant s from which to download files

4.2 Generate a random string r

4.3 Send a challenge message to s

4.4 Receive a response message from s containing its public key PKs and the challenge
    signed with its private key SKs

4.5 Download the files

4.6 Update experience and credibility repository

PEERS

Q.1 Upon receiving a search request (query message), check if any locally stored files
    match the query and if so send a queryOffer message

Q.2 Broadcast the query through the P2P network

P.1 Upon receiving a poll request (poll message), check if know any of the servers listed in
    the poll request and express an opinion on them by sending a pollReply message

P.2 Broadcast the poll request through the P2P network

P.3 Upon receiving an areYou message confirm the identity by sending an areYouReply

---

Fig. 1. Sequence of messages and operations in the P2PRep protocol
Figure 1 illustrates the operations executed and the messages exchanged in the context of a P2PRep enhanced interaction. In the figure, a “(G)” at the beginning of a step indicates that the step pertains to traditional Gnutella interchange; unmarked steps are peculiar to our protocol. The application of encryption is indicated with notations $[text]_{SK}$; when text is signed with private key $SK$, and $[text]_{PK}$, when text is encrypted with public key $PK$.

From the point of view of the servant looking for the resource, the protocol can be seen separated in four phases: 1) resource searching, 2) polling, 3) vote evaluation, and 4) resource downloading. Resource searching works like in a traditional Gnutella interchange. Then, in phase 2, $p$ determines, based on some preference criteria, a top list of offerers and broadcasts a Poll message requesting peers to vote on them. In the message, $p$ also includes a public key $PK_{poll}$ with which voters are requested to encrypt their replies to the poll (so to maintain the confidentiality of the votes when transiting on the network). Before encrypting their votes with $PK_{poll}$ for transmission, voters will sign them with their secret key. This allows $p$ to assess that votes have not been modified in transit. Phase 3 evaluates collected votes identifying possible cliques under the control of individual servants, discards suspicious voters, and selects a random set of voters which are contacted directly, via the pair (IP,port) they declared, to assess the correct origin of votes. Based on the election outcome, $p$ can decide the servant from which download the resources. The actual download is preceded by a challenge-response handshake between $p$ and the selected servant. In this way $p$ makes sure that it is actually talking to the servant corresponding to the declared identity. (The challenge-response exchange exploits the fact that a servant’s identifier is a digest of its public key). After the download completes, and based on its outcome, $p$ updates its local repositories where it maintains its own view of peers’ reputation and credibility.

Figure 5 also reports the operations of the other peers in the network. Peers, beside flooding the messages to others according to the Gnutella protocol, can originate responses by responding to queries as resource offerers (step Q.1) or to polls as voters (steps P.1 and P.2).

P2PRep was designed to be robust against most well-known security threats posed to reputation-based systems on anonymous P2P networks. Pseudo-spoofing and shilling are two typical attacks to reputations, both exploiting the fact that on a P2P network peers’ identities are not certified by any global authority. In pseudo-spoofing attacks, a malicious peer alleges that many witnesses are ready to vouch for its reputation, and produces a false witness list by forging the witness identities. In shilling attacks, a malicious peer actually creates and activates a number of false witnesses, ready to vouch for it. Recently, the term sybil attacks[?] has been proposed to designate all attacks of this kind. Our approach does not try to prevent all sybil attacks completely; rather, P2PRep tries to prevent pseudo-spoofing and increase the cost of shilling. Pseudo-spoofing prevention is obtained by checking the IP addresses of part of the peers that voted in favour of a servant: if a voter is not on line, its vote is eliminated from the poll. On the other hand, the choice of the IP addresses to be checked is not random,
but relies on an evaluation of their heterogeneity [20]. While our technique can indeed be beaten, doing so would require a malicious peer to incur in the cost of setting up false witnesses having highly heterogeneous IP addresses. Increases in IP address reliability may make this task potentially very difficult.

3 Design and implementation of a P2PRep enhanced servant

Component-based techniques are widely used for designing and implementing distributed software systems. We relied on component-based design both to reverse engineer and complement the internal structure of a Gnutella servant to support reputations.

3.1 P2PRep Extensions to Gnutella servants

Figure 2 shows the modular structure of a Gnutella servant; the dotted line encloses three additional components that are not present in ordinary servants but are needed to support the P2PRep protocol.

- The Reputation Manager manages all the new messages needed to send and receive reputation values.
- The Experience DB manages a repository (see Section 3.2) storing the experience values accumulated during past interactions with respect to reputations and credibilities.
- The Crypto Agent implements all encryption functions used in the protocol. This component must generate the key pair used in the asymmetric protocol.\(^5\)

\(^5\) In our current prototype we use RSA with 512 bits keys. This key can be encoded in 128 bytes, substituting 8 bits with two bytes in the set: \((0 - 9, A - F)\). This
Besides adding the auxiliary software components listed above, some minor changes to existing components must be introduced in order to manage reputation-related messages. Message routing in itself requires no changes; reputation-related messages can be routed as usual, because they are piggybacked on currently defined messages (Section 3.3). However, the part of GRouter that unpacks incoming messages should now recognize new unicast and multicast messages. This modification can be done quite easily in most current implementations.

The method invocations’ sequence of a file search is summarized in Figure 3. The User Manager sends a Query message to the GRouter containing the search criteria. The GRouter recognizes the incoming message as a multicast one and spreads it on the Gnutella network. When replies arrive, a standard Gnutella servant would pass them on to the User Manager component, which is responsible to show them to the user. In our prototype, GRouter selects a subset of the servants listed in the reply set; choice criteria are the servants’ connection speeds, the extra information included in some version of the protocol,6 and the content of Experience DB, which is queried using the servant identifier ServentID as key. The Reputation Manager uses the record extracted from Experience DB to decide whether to trust the corresponding servant or not. After a set of reliable servants has been chosen, a poll message is broadcast by the GRouter. Replies to the poll request are forwarded to the Reputation Manager. The last

---

5 Substitution is done whenever the field is a null-terminated string. The MD5 digest algorithm is used, producing signatures of 128 bits, encodable in 16 or 32 bytes.

6 For instance OpenData [20] allows servant to store the speed of their last 10 uploads, if they have successfully uploaded at least one file, if they are busy, and also if their connection is firewalled.
step of P2PRep requires this component to verify votes through a direct connection. The percentage of servants that are actually contacted is a configuration parameter that is set by the user.

Once verification has been completed, the Resource Locator can associate, with each servant, the sum of all positive votes about it.7 Votes given by servants that are considered not trustable by the credibility repository are ignored, and the best servant is chosen for download via HTTP Client/Server.

After downloading, the user may cast a vote about the resource. This binary vote is used to update Experience DB, and properly changing the reputation of the servant that provided the resource and those who voted for it.

At the receiving end, the method invocation sequence is summarized in Figure 4. Incoming queries are unpacked by GRouter and sent to Resource Locator which is responsible of checking in the file repository if there are files that match that criteria. The information on these files is sent to GRouter that just send it back, in a QueryHit message. When the corresponding Poll reply arrives, it is sent to the Reputation Manager which checks if there are records about the servants listed in the message. If this is the case, it passes the encrypted information about them to the GRouter. The latter then sends a PollHit message. Note that IP verification messages come directly to the HTTP Client/Server component, that is able to reply correctly.

### 3.2 Repository Schema

Several schema and data model solutions can be adopted for the reputations' repository managed by the Experience DB component. In our current implementa-

---

7 Our current implementation considers binary votes, where 1 represents a positive vote (in favor of the servant) and 0 a negative vote.
tation, the reputations' repository consists of two tables, namely **experience repository** and **credibility repository**. The digest of each servant's public key is used to compute the ServentID, which is the primary key of both tables. The tables' schemata are summarized below:

- **experience repository**: (ServentID, num_plus, num_minus, timestamp)
- **credibility repository**: (ServentID, num_agree, num_notagree, timestamp)

In the above schemata, num_plus and num_minus respectively store the servant's positive and negative experiences with the peer having servant identifier ServentID, while num_agree and num_notagree contain the number of times that the vote expressed by ServentID was confirmed or not by the servant's experience. ServentID is 16 bytes long while the remaining fields are 4 bytes each. The purpose of the timestamp is supporting an experience and credibility lifecycle, allowing for filtering out entries unused for a long time. Each entry of the experience table is 28 bytes long, this means that less than 700Kb are needed for 25,000 servents; therefore, a relatively limited amount of storage should be enough to keep trace of both experience and credibility of all relevant servents. It is worth noticing that P2P applications are typically used to exchange resources of considerable size and the space required, even by an extensive repository, should be quite manageable.

### 3.3 Additional protocol messages

The implementation of P2PRep requires two additional protocol messages: Poll and PollHit. Since interoperability with the existing Gnutella network is of paramount importance, new messages introduced by our extension should be routable via standard servents. Our implementation satisfies these requirements by piggybacking the additional messages on standard Query and QueryHit Gnutella messages. Namely, Poll messages are implemented using the field search criteria of standard Query messages. As described before, Poll messages carry a servant list and a session public key. In order to identify the message and its encoding correctly, the payload starts with the string REP:poll:HEX as shown in Figure 5, followed by a list of the ServentIDs of all servents whose reputation is being checked. At the end of the message there is the public key of the polling session. Standard Gnutella servents will process piggybacked queries as ordinary ones that do not match any file. In turn, PollHit messages are realized piggybacking QueryHit Gnutella messages. The first file entry contains zeros both as index and size, and the string REP:phit:HEX is specified as filename. We use the private data field in the trailer (introduced by version v1.3.0 of the BearShare servant), to store the encrypted payload and its signature. The payload, encrypted with RSA(512), begins with the servant's IP and Port. A sequence of index-vote pairs follows, where index is a two bytes field specifying the position of the ServentID in the Poll message, while vote is encoded in one byte. The payload ends with the Servent public key. A 64 bytes long signature for the whole message is appended. It is computed as follows: a digest of the plaintext
payload (MD5) is encrypted (using RSA) with the private key associated to the servent public key. Finally, some of the incoming PollHit messages are chosen for vote checking and IP addresses listed in the messages are checked via a direct connection. The message used to this purpose has the form AreYou(ServentID), where ServentID is encoded in 32 bytes.

### 3.4 Performance

The P2Prep protocol requires a greater number of message exchanges for every successful request. A limited impact on the performance of the P2P network is a critical success factor for the protocol. The protocol additional exchanges can be distinguished in broadcast and direct messages. It is reasonable to assume that the major load of the protocol on network performance is due to broadcast messages. Direct communication requires the exchange of a limited number of quick
messages. For instance direct connections are used to implement the AreYou message, which is a call to the HTTP Client/Server module. This exchange is very quick, as the message contains only few bytes and it is directed to a few of the nodes that expressed their votes on a servant. Indeed, most performance models of P2P networks stress as a limiting factor the aggregate bandwidth required by the exchange of broadcast messages.

To evaluate the impact that the broadcast Poll messages could have on the P2P network, we ran a few experiments that returned a few quantitative measures, on which a preliminary analysis was based.

A first series of experiments was dedicated to identify the regularity on the behavior of the network, in terms of number of reachable hosts and number of answers to a query.

The experiments were realized connecting our client to 10 fast servents. Each session lasted 10 minutes and at the end of the session we recorded the number of hosts that were signaled as reachable by the protocol. We repeated this experiment 100 times in two days and the results are shown in Table 1. After 5 minutes in each session we started a search for a common file (5 minutes are in our experience sufficient to reach a stable horizon, with a fast Internet connection as the one we were using). The node waited for results of the search for 5 minutes, until the end of the session.

The results we obtained indicate a relative stability of the network configuration. Both the number of reachable hosts and the number of replies to the search request exhibit a limited variation interval for the 95% of the measures. Also, the number of hosts that are reachable is relatively high and this guarantees that a sizable portion of the Gnutella network is within the horizon. It is then reasonable to assume that the protocol will be able to find an adequate reputation support for the nodes which have offered, for a period, quality resources on the network.

We now estimate the size of the messages required to evaluate the reputation of servents offering a resource. First, we observe that many servents can be polled with a single Poll message. The size of Poll message is proportional to the number of servents to inquire. As most implementations drop messages

<table>
<thead>
<tr>
<th># of responders</th>
<th># of reachable servents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>785.7</td>
</tr>
<tr>
<td>Standard Error</td>
<td>43.0</td>
</tr>
<tr>
<td>Median</td>
<td>679</td>
</tr>
<tr>
<td>Range</td>
<td>2213</td>
</tr>
<tr>
<td>Minimum</td>
<td>71</td>
</tr>
<tr>
<td>Maximum</td>
<td>2284</td>
</tr>
<tr>
<td>95% CI min</td>
<td>696.3</td>
</tr>
<tr>
<td>95% CI max</td>
<td>875.1</td>
</tr>
</tbody>
</table>

Table 1. Statistical measures obtained by 100 experimental sessions
<table>
<thead>
<tr>
<th>Class</th>
<th>Speed</th>
<th># of servants</th>
<th>% over total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell</td>
<td>0</td>
<td>361,804</td>
<td>3.32%</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>55,293</td>
<td>0.51%</td>
</tr>
<tr>
<td>28</td>
<td>28</td>
<td>715,546</td>
<td>6.56%</td>
</tr>
<tr>
<td>56K</td>
<td>53</td>
<td>460,544</td>
<td>4.22%</td>
</tr>
<tr>
<td>ISDN</td>
<td>128</td>
<td>1,665,283</td>
<td>15.27%</td>
</tr>
<tr>
<td>Cable</td>
<td>384</td>
<td>3,351,010</td>
<td>30.73%</td>
</tr>
<tr>
<td>DSL</td>
<td>768</td>
<td>1,151,967</td>
<td>10.56%</td>
</tr>
<tr>
<td>T1</td>
<td>1500</td>
<td>1,508,287</td>
<td>13.83%</td>
</tr>
<tr>
<td>T3</td>
<td>&gt;1500</td>
<td>1,634,979</td>
<td>14.99%</td>
</tr>
<tr>
<td>Total</td>
<td>10,904,713</td>
<td>100%</td>
<td></td>
</tr>
</tbody>
</table>

Table 2. Speed of servants that responded to queries

bigger than 64k bytes [19], a Poll message can carry up to around 4,000 different ServentID. We assume that it will not be necessary to ask on the reputation of so many servants. Instead, we assume that when a resource is offered by many servants, the client could select a subset of the servants to inquire, using as selection parameters the bandwidth that servants offer (faster is better) and heterogeneity in the IP address. Assuming to choose only 10 to 20 servants, the impact of the polling phase becomes comparable to an additional query delivered onto the network. To evaluate the distribution of network bandwidth among the servants, we ran another series of experiments. For a total of 32 hours a Gnutella node logged all the QueryHit traffic. The speed of the servants is distributed as shown in Table 2. We can see that most of the servants declare a high bandwidth, it is then reasonable to assume that a restriction only to servants offering a high bandwidth will not pose a limit on the number of servants that could offer the resource.

Several optimizations can reduce the impact of P2PRep on network performance. For instance, reputations can be cached on the nodes, and servants that have already been voted as reliable following a search, can keep the reputation for the remainder of the session. Reputations may be kept across sessions, further reducing the number of polling requests (at the expense of an increase in the storage requirements and a decrease in the responsiveness of the network to node misbehavior).

4 Handling resource-based reputation

The solution described in this paper considered reputations associated with servants. An alternative (or complementary) solution consists in associating reputations with resources themselves. Intuitively, each resource can be associated with a digest computed applying a secure hash function to the resource’s content. When a servant downloads a resource, it can record whether the resource (identified by its digest) is satisfactory or not. Then, it can share this opinion
Fig. 6. Extensions to Gnutella messages in XP2PRep

with others by responding to Poll requests broadcasted by a protocol’s initiator to enquire about resource reputations (in contrast to servent’s reputations). With respect to message exchanges, the resource-based reputation solution works essentially in the same way as the servent-based one. The messages content is adapted to support references to resources as illustrated in Figure 6.

Also, experience repository will need to refer to resources. Unlike with servents it is sufficient to maintain just a record stating whether the resource is reliable or not (there is no need to maintain good and bad records). The repository has then three fields: digest, vote, and timestamp. With a digest 16-bytes long, a vote 1-byte long, and timestamp 4-bytes long, we can store up to 50,000 entries in 1 Mb. Therefore, while the number of resources is expected to be much greater than the number of servents, the storage requirements appear not to be a problem.

5 Conclusions and Future Work

We have presented the design and implementation of a reputation-aware Gnutella servent. The overall rationale and some specific choices of our design were discussed. Our experience shows that introducing reputations does not require extensive re-engineering either of existing clients or of the Gnutella protocol itself. The additional performance burden of managing and exchanging reputation data does not seem to significantly degrade Gnutella’s performance. The piggybacking technique adopted in our implementation allows our reputation-aware servents
to take advantage of the evolution of the standard Gnutella protocol. In our opinion, P2PRep performance impact is potentially well counterbalanced by the overall increase in P2P network security that could result from its large scale adoption.

In the long run, reputation-based protocols may even allow P2P systems to preserve anonymity without the need of costly central agencies for managing identities [7]. We are well aware that servant self-regulation alone cannot guarantee identity persistence; however, cooperative solutions like P2PRep encourage servants that want to act as distribution points to keep a single, trusted identity.

P2P applications are evolving very quickly and many of the results we present in the paper will have to be adapted to the architectures that are now going to be implemented (e.g., architectures with ultra-peers, as the architecture on which the last version of the Gnutella protocol is based). Also, our future implementations will rely on richer query support provided by sophisticated Gnutella servants. Limewire, for instance, implements a new XML-based encoding for Gnutella queries. With respect to Gnutella 0.4 queries [20], the Limewire format uses an additional section, located after the ordinary query, to contain the extended query. A valid extended query has to comply with an XML schema specified via a Uniform Resource Identifier (URI). Several standard URIs have already been defined, but other XML schemata can be added. By defining a suitable XML schema, meta-information could be added to our protocol messages, enabling P2PRep aware servants to exchange semantically richer information about other servants and resources.

References


---

8 For example, Limewire's Ping messages are not routed as usual multicast messages, but cached. This solution could be used also for ServenID, fostering reduction of the length of PollHit messages.


