Implementing a Reputation-Aware Gnutella Servent

Fabrizio Cornelli[1], Ernesto Damiani[1], Sabrina De Capitani di Vimercati[2], Stefano Paraboschi[3], and Pierangela Samarati[1]

[1] Dipartimento di Tecnologie dell'Informazione - Università di Milano
Via Bramante 65 - 26013 Crema (CR) - Italy
fcornelli@crema.unimi.it,{damiani,samarati}@dti.unimi.it

[2] Dipartimento di Elettronica per l'Automazione - Università di Brescia
Via Branze 38 - 25123 Brescia (BS) - Italy
decapita@ing.unibs.it

[3] Dipartimento di Elettronica e Informazione - Politecnico di Milano
Via Ponzio 34/5 - 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 [4, 6–11, 17]. 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 [2, 6, 12, 13]. 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 [1]. 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 [3] 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* [10] 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 [3] 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 servent p looking for a resource broadcasts to all its neighbors a `Query` message reporting the search keywords. Each servent 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 *servent_id* of the responder, a *trailer* with application-specific data, and the pair $\langle$IP,port$\rangle$ 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 servent 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 servent downloads a resource from another servent, it can record whether or not the download has completed to its satisfaction. Before downloading a resource from another servent, a servent can poll its peers about their knowledge of the offerer, thus assessing its reputation. If that offerer is then chosen, the downloading servent 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 servent's reputation and credibility, identifiers' persistency must be assumed. In traditional Gnutella, the identifier with which a servent presents itself can, in principle, change at every session. However, a servent wishing to establish some reputation as offerer or voter is encouraged to maintain its identifier persistent. Note that this does not imply that the servent'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 servent's identifier is assumed to be the digest of a public key, obtained using a secure hash function, and for which the servent 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 servent's identifier or can be generated ad-hoc for the exchange (if the identifier of the involved servent 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`(min_speed,$search\_string$)
(**G**) 1.2 Receive a set of offers from offerers O
`QueryHit`(num_hits,$port$,IP,$speed$,$Result$,$trailer$,$servent\_id_i$)
Phase 2: Polling
2.1 Select top list $T \subseteq O$ of offerers
2.2 Generate a pair of public, secret keys (PK_{poll},SK_{poll})
2.3 Poll peers about the reputations of offerers T
`Poll`(T,PK_{poll})
2.4 Receive a set of votes from voters V
`PollReply`($\{([(IP, port, Votes, servent\_id_i)]_{\mathrm{SK}_i}, \mathrm{PK}_i)\}_{\mathrm{PK}_{poll}}$)
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`($servent\_id_j$)
3.3 Expect back confirmation messages from each selected voter
`AreYouReply`($response$)
Phase 4: Resource downloading
4.1 Select servent s from which download files
4.2 Generate a random string r
4.3 Send a `challenge` message to s
`challenge`(r)
4.4 Receive a `response` message from s containing its public key PK_s and the challenge signed with its private key SK_s
`response`($[r]_{\mathrm{SK}_s}$,PK_s)
4.5 If the challenge-response exchange fails terminate the process
(**G**) 4.6 Download the files
4.7 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 `QueryHit` 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 servents 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 $[text]_{\mathrm{SK}}$, when *text* is signed with private key SK, and $\{text\}_{\mathrm{PK}}$, when *text* is encrypted with public key PK.

From the point of view of the servent looking for the resource, the protocol can be 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 the authenticity of the votes and also that they have not been modified in transit. Phase 3 evaluates collected votes identifying possible cliques under the control of individual servents, discards suspicious votes, and selects a random set of voters which are contacted directly, via the pair $\langle$IP,port$\rangle$ they declared, to assess the correct origin of votes. Based on the election outcome, p can decide the servent from which download the resources. The actual download is preceded by a challenge-response handshake between p and the selected servent. In this way p makes sure that it is actually talking to the servent corresponding to the declared identity. (The challenge-response exchange exploits the fact that a servent'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 1 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. *Pseudospoofing* and *shilling* are two typical attacks to reputations [10], 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* [5] 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 pseudospoofing and increase the cost of shilling. Pseudospoofing prevention is obtained by checking the IP addresses of part of the peers that voted in favor of a servent: 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 [16]. 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 servent

Component-based techniques are widely used for designing and implementing distributed software systems. We relied on component-based design both to re-

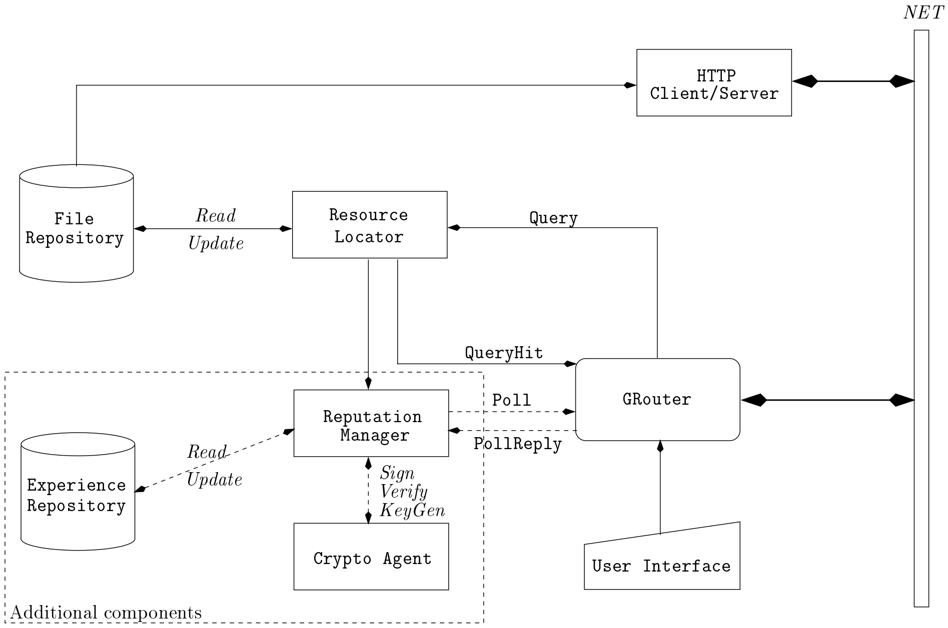

Fig. 2. Typical component-based structure of a Gnutella servent

verse engineer and complement the internal structure of a Gnutella servent to support reputations.

3.1 P2PRep Extensions to Gnutella servents

Figure 2 shows the modular structure of a Gnutella servent; the dotted line encloses three additional components that are not present in ordinary servents 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 Repository` is 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 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.

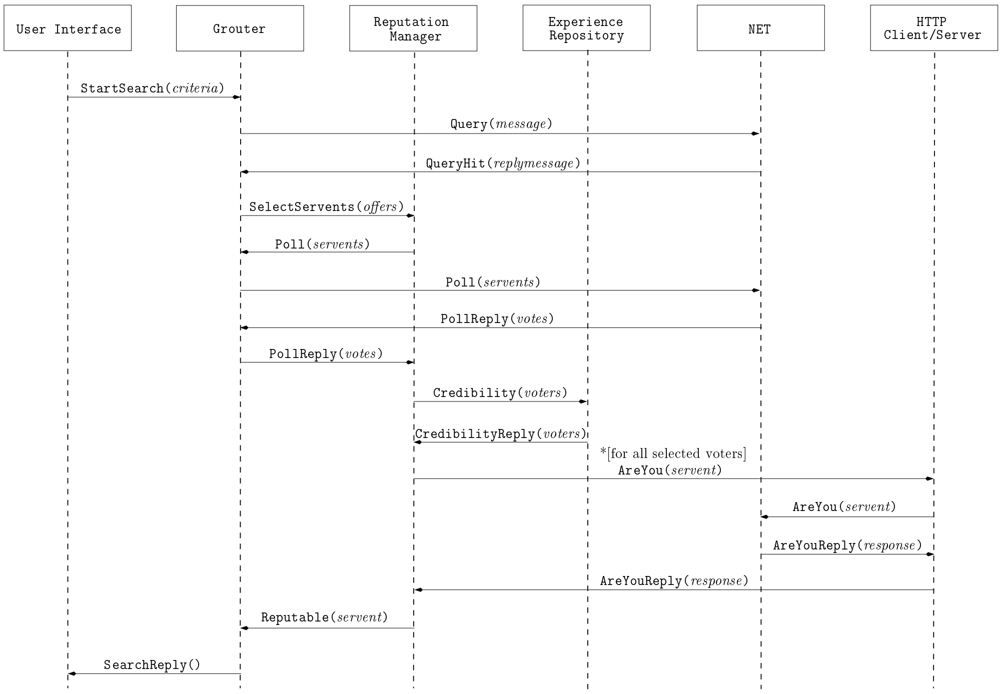

Fig. 3. UML Sequence diagram of a search session

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 Interface` 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 servent would pass them on to the `User Interface` component, which is responsible to show them to the user. In our prototype, the `Reputation Manager` selects a subset of the servents listed in the reply set; choice criteria are the servents' connection speeds, the extra information included in some version of the protocol,[6] and, possibly, the content of `Experience Repository`, which is queried

[6] For instance, the `open data` field [16] allows servent to specify if they have successfully uploaded at least one file, if the servent's upload slots are currently full, and also if their connection is firewalled.

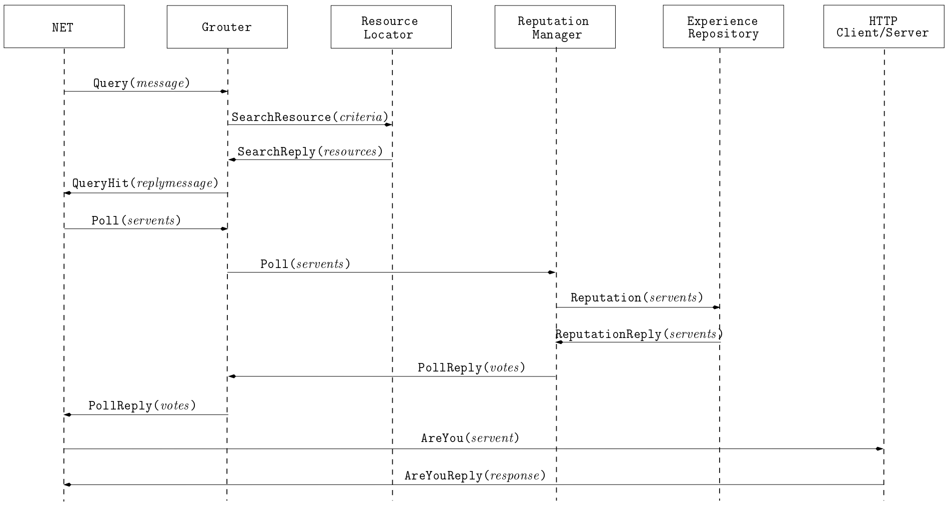

Fig. 4. UML Sequence diagram of a reply session

using the servent identifier *servent_id* as key. In such a case, the `Reputation Manager` uses the record extracted from the `Experience Repository` to decide whether to trust the corresponding servent or not. After a set of reliable servents has been chosen, a poll message is broadcast by the `GRouter`. Replies to the poll request are forwarded to the `Reputation Manager`. The last step of P2PRep requires this component to verify votes through a direct connection. The percentage of servents that are actually contacted is a configuration parameter that is set by the user.

Once verification has been completed, the `Reputation Manager` can associate, with each servent, the sum of all positive votes about it.[7] Votes given by servents that are considered not trustable according to the credibility information stored in the `Experience Repository`are ignored, and the best servent 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 Repository`, and properly changing the reputation of the servent 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 the `GRouter` and sent to the `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 the `GRouter` that just send it back, in a `QueryHit` message. When a `Poll` message arrives, it is sent to the `Reputation Manager` which checks if the `Experience Repository`includes information about the servents listed in the message. If this

[7] Our current implementation considers binary votes, where `1` represents a positive vote (in favor of the servent) and `0` a negative vote.

is the case, it passes the encrypted information about them to the `GRouter`. The latter then sends a `PollReply` message. Note that IP verification messages (`AreYou` 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 reputation and credibility information managed by the `Experience Repository` component. In our current implementation, the `Experience Repository` consists of two tables, namely `reputation_repository` and `credibility_repository`. The tables' schemata are summarized below:

- `reputation_repository`: (*servent_id*,*num_plus*,*num_minus*,*timestamp*)
- `credibility_repository`: (*servent_id*,*num_agree*,*num_notagree*,*timestamp*)

In the above schemata, *servent_id* is the primary key of both tables, *num_plus* and *num_minus*, respectively, store the servent's positive and negative experiences with the peer having servent identifier *servent_id*, while *num_agree* and *num_notagree* contain the number of times that the vote expressed by *servent_id* was confirmed or not by the servent's experience. *servent_id* 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 `PollReply`. 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 this requirement 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 servent 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 *servent_ids* 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, `PollReply` messages are realized piggybacking `QueryHit` Gnutella messages. The first file entry contains

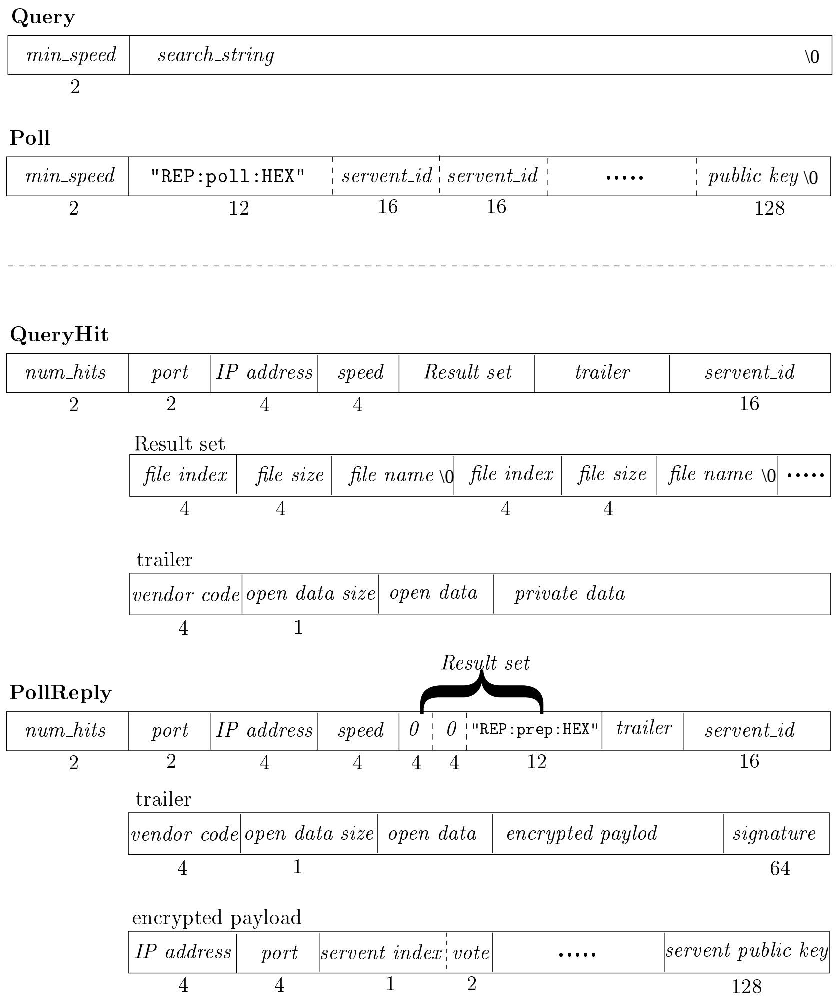

Fig. 5. Extensions to Gnutella messages in `P2PRep`

zeros both as index and size, and the string `REP:prep:HEX` is specified as filename. We use the *private data* field in the trailer (introduced by version v1.3.0 of the `BearShare` servent), to store the encrypted payload and its signature. The payload, encrypted with RSA(512), begins with the servent's `IP` and `Port`. A sequence of `index-vote` pairs follows, where `index` is a two bytes field specifying the position of the *servent_id* 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 plain text payload (MD5) is encrypted (using RSA) with the private key associated to the servent public key. Finally, some of the incoming `PollReply` messages are chosen for vote checking and IP addresses listed in the messages

	# of responders	# of reachable servents
Mean	785.7	343474.2
Standard Error	45.0	21095.6
Median	679	342000
Range	2213	946000
Minimum	71	16000
Maximum	2284	962000
95% CI min	696.3	301599.7
95% CI max	875.1	385348.7

Table 1. Statistical measures obtained by 100 experimental sessions

are checked via a direct connection. The message used to this purpose has the form `AreYou`*servent_id*, where *servent_id* is encoded in 16 bytes.

3.4 Performance

The P2PRep protocol requires additional 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 servent. 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

Class	Speed	# of servents	% over total
Cell	0	361,804	3.32%
14	14	55,293	0.51%
28	28	715,546	6.56%
56K	53	460,544	4.22%
ISDN	128	1,665,283	15.27%
Cable	384	3,351,010	30.73%
DSL	768	1,151,967	10.56%
T1	1500	1,508,287	13.83%
T3	>1500	1,634,979	14.99%
Total		**10,904,713**	**100%**

Table 2. Speed of servents that responded to queries

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 a `Poll` message is proportional to the number of servents to inquire. As most implementations drop messages bigger than 64Kb [15], a `Poll` message can carry up to around 4,000 different *servent_id*. We assume that it will not be necessary to ask on the reputation of so many servents. Instead, we assume that when a resource is offered by many servents, the client could select a subset of the servents to inquire, using as selection parameters the bandwidth that servents offer (faster is better) and heterogeneity in the IP address. Assuming to choose only 10 to 20 servents, 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 servents, we ran another series of experiments. For a total of 32 hours a Gnutella node logged all the `QueryHit` traffic. The speed of the servents is distributed as shown in Table 2. We can see that most of the servents declare a high bandwidth, it is then reasonable to assume that a restriction only to servents offering a high bandwidth will not pose a limit on the number of servents 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 servents 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

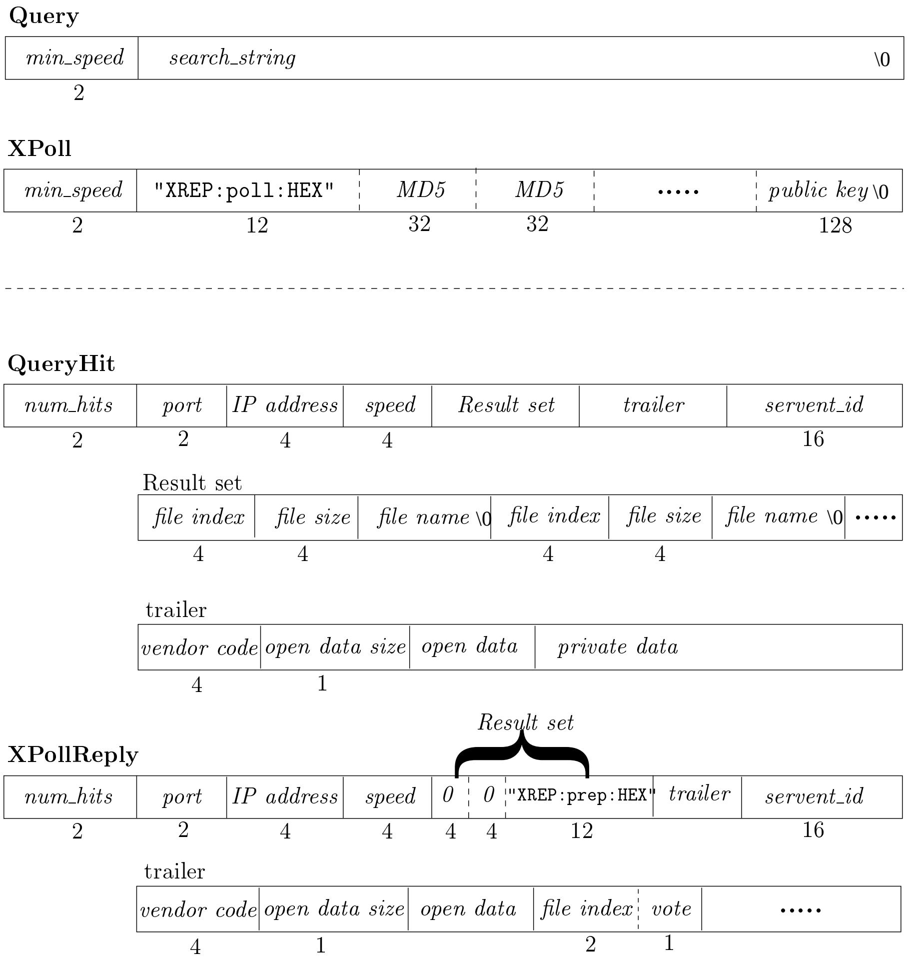

Fig. 6. Extensions to Gnutella messages in XP2PRep

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 servents. 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 servent downloads a resource, it can record whether the resource (identified by its digest) is satisfactory or not. Then, it can share this opinion with others by responding to `Poll` requests broadcasted by a protocol's initiator to inquire 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.[8] 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 [5]. We are well aware that servent self-regulation alone cannot guarantee identity persistence; however, cooperative solutions like P2PRep encourage servents 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 servents. `LimeWire`, for instance, implements a new XML-based encoding for Gnutella queries [14]. With respect to Gnutella 0.4 queries [16], 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

[8] For example, `LimeWire`'s `Ping` messages are not routed as usual multicast messages, but cached. This solution could be used also for *servent_id*, fostering reduction of the length of `PollReply` messages.

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 servents to exchange semantically richer information about other servents and resources.

References

1. S. Bellovin. Security aspects of Napster and Gnutella. In *Proc. of USENIX 2001*, Boston, June 2001.
2. I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proc. of the ICSI Workshop on Design Issues in Anonymity and Unobservability*, Berkeley, CA, July 2000.
3. F. Cornelli, E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. Choosing reputable servents in a P2P network. In *Proc. of the Eleventh International World Wide Web Conference*, Honolulu, Hawaii, May 2002.
4. R. Dingledine, M.J. Freedman, and D. Molnar. The Free Haven project: Distributed anonymous storage service. In *Proc. of the Workshop on Design Issues in Anonymity and Unobservability*, Berkeley, California, USA, July 2000.
5. J.R. Douceur. The sybil attack. In *Proc. of the IPTPS02 Workshop*, Cambridge, MA (USA), March 2002.
6. P. Druschel and A. Rowstron. Past: A large-scale persistent peer-to-peer storage utility. In *Proc. of the Eight IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, Schoss Elmau, Germany, May 2001.
7. P. Golle and K. Leyton-Brown. Incentives for sharing in peer-to-peer networks. In *Proc. of the Third ACM Conference on Electronic Commerce*, Tampa, Florida, USA, October 2001.
8. L. Gong. JXTA: A network programming environment. *IEEE Internet Computing*, 5(3):88–95, May/June 2001.
9. Openprivacy. http://www.openprivacy.org.
10. A. Oram, editor. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies.* O'Reilly & Associates, March 2001.
11. M. Parameswaran, A. Susarla, and A.B. Whinston. P2P networking: An information-sharing alternative. *IEEE Computer*, 34(7):31–38, July 2001.
12. M. Ripeanu. Peer-to-peer architecture case study: Gnutella network. Technical Report TR-2001-26, University of Chicago, Department of Computer Science, July 2001.
13. S. Saroiu, P.K. Gummadi, and S.D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proc. of the Multimedia Computing and Networking*, San Jose, CA, January 2002.
14. S. Thadani. Meta information searches on the Gnutella network. http://rfc-gnutella.sourceforge.net/Proposals/MetaData/meta_information_searches.htm.
15. S. Thadani. Free riding on Gnutella. Technical report, LimeWire LLC, 2001. http://www.limewire.org.
16. *The Gnutella Protocol Specification v0.4 (Document Revision 1.2)*, June 2001. http://www.clip2.com/GnutellaProtocol04.pdf.
17. B. Yang and H. Garcia-Molina. Comparing hybrid peer-to-peer systems. In *Proc. of the 27th International Conference on Very Large Data Bases*, Rome, Italy, September 2001.