A Protocol for Reputation Management in Super-Peer Networks

Shalendra Chhabra
University of California, Riverside
California, USA
schhabra@cs.ucr.edu

Ernesto Damiani
Università di Milano
26013 Crema - Italy
damiani@dti.unimi.it

Sabrina De Capitani di Vimercati
Università di Milano
26013 Crema - Italy
decapita@dti.unimi.it

Stefano Paraboschi
Università di Bergamo
24044 Dalmine - Italy
parabosc@unibg.it

Pierangela Samarati
Università di Milano
26013 Crema - Italy
samarati@dti.unimi.it

Abstract

Peer-to-Peer (P2P) applications have recently seen an enormous success and have reached millions of users. The main reason of this success is the anonymity the users enjoy. However, as recent experiences with P2P networks show, this anonymity offers an opportunity to exploit the network for abuses (e.g., the spread of malware).

In this paper we extend our previous work on P2PRep, a reputation management protocol for pure P2P networks, in the case of super-peer networks. We present the design and implementation of reputation-aware servents.

1. Introduction

In this paper we extend our previous work on the reputation management protocol P2PRep [3]. We propose the *SupRep* protocol and present its design and implementation. We have developed the SupRep protocol on the top of Gnutella 0.6 [5] which is the standard of Gnutella at the time of writing of this paper. We have chosen Gnutella 0.6 because it has a distributed architecture with super-peers (Ultrapeers)[1] for file exchange and its specifications are open and it is one of the most widely used protocol. We introduce the role of *repeater* assigned to selected peers (and super-peers) to support interaction among servents behind firewalls. We also study the case of malicious super-peers and show that the protocol is robust to known attacks. Due to the strict page limit, we will assume that the reader is familiar with the concept of super-peers in P2P networks [8].

1 Ultrapeers, super-peers and super-nodes are terms used interchangeably.

2. SupRep Protocol

Each servent has associated a self-appointed *serventID*, which can be communicated to others when interacting, as established by the P2P communication protocol used. The *serventID* of a party (intuitively a user connected at a machine) can change at any instantiation or remain persistent. However, persistence of a *serventID* does not affect anonymity of the party behind it, as the *serventID* works only as an opaque identifier.[2] Our approach encourages persistence as the only way to maintain history of a *serventID* across transactions.

In a Gnutella-like environment, a servent p looking for a resource broadcasts a Query message, and selects, among the servents responding to it (which we call *offerers*), the one from which to execute the download. Our approach is to allow p, before deciding from where to download the resource, to inquire about the reputation of offerers by polling its peers. The basic idea is as follows. After receiving the responses to its query, p can select a servent, or a set of servents. Then, p polls its peers by broadcasting a message PollRequest requesting their opinion about the selected servents. All peers can respond to the poll with their opinions in PollReply about the reputation of each of such servents. The super-peers in the network collect the PollReply messages coming from the leaf nodes in their own cluster, considering a certain timeout and synthesize these messages into one single CumulativePollReply message containing the encrypted leaf votes (which it collected from individual PollReply as mentioned above), and its own encrypted votes (if it wants to express them). The poll requestor p parses the CumulativePollReply message and extracts the

2 It must be noted that, while not compromising anonymity, persistent identifiers introduce linkability, meaning transactions coming from the same servent can be related to each other.

Proceedings of the 15th International Workshop on Database and Expert Systems Applications (DEXA'04)
1529-4188/04 $ 20.00 IEEE

votes. The votes are then checked (we will elaborate more on this in Section 3) and, once validated, the poll requestor p can use the opinions expressed by these *voters* to make its decision. Note that voters declare their *serventID*, which can then be taken into account by p in weighing the votes received (p can judge some voters as being more credible than others).

The intuition behind our approach is therefore very simple. A little complication is introduced by the need to prevent exposure of polling to security violations by malicious parties. In particular, we need to ensure authenticity of servents acting as offerers or voters (i.e., preventing impersonation) and the quality of the poll. Ensuring the quality of the poll means ensuring the integrity of each single vote (e.g., detecting modifications to votes in transit) and rule out the possibility of dummy votes expressed by servents acting as a clique under the control of a single malicious party. In Section 2.2 we describe how these issues are addressed in our protocol.

2.1. Notations in SupRep

Our protocol assumes the use of *public key* encryption to provide integrity and confidentiality of message exchanges. Whether permanent or fresh at each interaction, we require each *serventID* to be a digest of a *public key*, obtained using a secure hash function and for which the servent knows the corresponding *private key*. This assumption allows a peer talking to a *serventID* to ensure that its counterpart knows the *private key*. A pair of keys is also generated on the fly for each poll. In the following we will use (PK_i,SK_i) to denote a pair of public and private keys associated with i, where i can be a servent or a PollRequest. We use $\{M\}_K$ to denote the encryption of a message M under key K. Also, in illustrating the protocol, we will use p to denote the protocol's initiator, $\mathcal{S}$ to denote the set of servents connected to the P2P network at the time p sends the Query, O to denote the subset of $\mathcal{S}$ responding to the Query (*offerers*), V to denote the subset of $\mathcal{S}$ responding to p's polling (*voters*), U to denote the set of UltraPeers and R to denote the set of *repeaters*. A message transmission from servent x to servent y via the P2P network will be represented as $x \longrightarrow y$, where "$*$" appears instead of y in the case of a broadcast transmission. A direct message transmission (outside the P2P network) from servent x to servent y will be represented as $x \xrightarrow{\text{D}} y$.

2.2. Working of the SupRep protocol

The polling protocol, illustrated in Figure 1, works as follows.

Like in the conventional Gnutella protocol, the servent p looking for a resource sends a Query indicating the resource it is looking for. Every servent receiving the Query and willing to offer the requested resource for download, sends back a QueryHit message stating how it satisfies the Query (i.e., number of files matching the query, the set of responses, and the speed in Kb/second) and providing its *serventID* and its pair ⟨IP,port⟩, which p can use for downloading. Then, p selects its top list of servents T and polls its peers about the reputations of these servents. In the PollRequest, p includes the set T of *serventID*s about which it is inquiring and a public key PK_{poll} generated on the fly for the PollRequest, with which responses to the poll will need to be encrypted.[3] The PollRequest is sent through the P2P network like the Query request and therefore p does not need to disclose its *serventID* or its IP to be able to receive back the response.

A servent receiving the PollRequest and wishing to express an opinion on any of the servents in T can do so by responding to the PollRequest with a PollReply message which contains the required information encrypted with the PK_{poll}. The encrypted payload contains the pair ⟨IP,port⟩, *serventID*, *public key* of the responding host, its votes, PK_{poll}, and a signature. Note that the key PK_{poll} in the PollReply message has the role of a "poll session identifier". In this way, a malicious peer cannot collect "old" PollReply messages and resend them in correspondence of a new polling. Super-peers in SupRep are assigned the responsibility to wait for a timeout for the PollReply messages coming from the leaf nodes in their cluster. The super-peers then synthesize a CumulativePollReply message containing the encrypted votes of their leafs for a corresponding PollRequest along with its own encrypted votes. The super-peers then forward this CumulativePollReply message to the poll requestor.

The fact that the votes are encrypted with PK_{poll} protects their confidentiality and allows the detection of integrity violations. Therefore, as a consequence of the poll, p receives a set of votes, where, for each servent in T, some votes can express a good opinion while some others can express a bad opinion.

To base its decision on the votes received, p needs to trust the reliability of the votes. Thus, p first uses decryption to detect tampered with votes and discards them. Second, p detects votes that appear suspicious, for example since they are coming from IPs suspected of representing a clique. Third, p selects a set of voters that it directly contacts (by using the ⟨IP,port⟩ pair they provided) to check whether they actually expressed that vote. For each selected voter v_j, p directly sends a TrueVote request reporting the votes it has received from v_j, and expects back a confirmation message TrueVoteReply from v_j confirming the validity of the vote

3 In principle, p's key could be used for this purpose, but this choice would disclose the fact that the request is coming from p.

Proceedings of the 15th International Workshop on Database and Expert Systems Applications (DEXA'04)
1529-4188/04 $ 20.00 IEEE

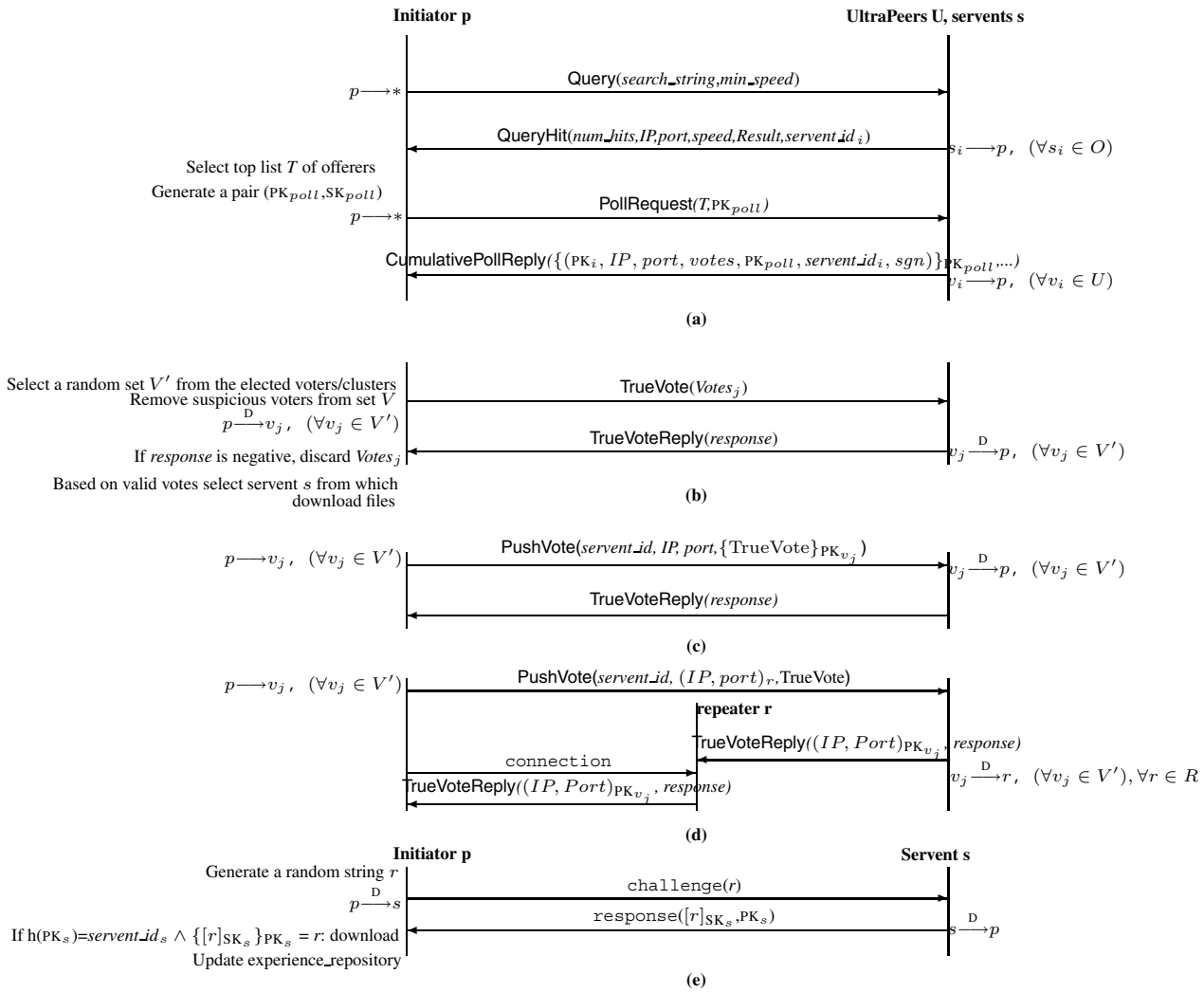

Figure 1. SupRep protocol: query and poll (a), vote verification (b)-(d), and resource download (e)

(see Figure 1(b)). If any of the poll requestor or poller is behind the firewall, then the push verification and repeater mechanism as described in Section 3 is followed. The vote verification mechanism forces potential malicious servents to pay the cost of using real IPs as false witnesses. Note that of course nothing forbids malicious servents to completely throw away the votes in transit (but if so, they could have done this blocking on the QueryHit in the first place). Also note that servents will not be able to selectively discard votes, as their recipient is not known and their content, being encrypted with PK_{poll} is not visible to them. Upon assessing correctness of the votes received, p can finally select the offerer it judges as its best choice according to *i*) connection speed, *ii*) its own reputation about the servents, *iii*) the reputation expressed in the votes received, and *iv)* the *credibility* associated with voters which it will use to properly weigh the votes they express when responding to a PollRequest.

At this point, before actually initiating the download, p challenges the selected offerer s to assess whether it corresponds to the declared *serventID*. Servent s will need to respond with a message containing its *public key* PK_s and the challenge signed with its *private key* SK_s. If the challenge-response exchange succeeds and the PK_s's digest corresponds to the *serventID* that s has declared, then p will know that it is actually talking to s. Note that the challenge-response exchange is done via direct communication, like the download, in order to prevent impersonation by which servents can offer resources using the *serventID* of other peers. With the authenticity of the counterpart established, p can initiate the download and, depending on its satisfaction for the operation, update the reputation associated with s and stored in a repository called *experience_repository*.

3. The vote verification mechanism and the role of *repeaters*

The vote verification mechanism is performed in two different ways depending on whether the poll requestor and the poller are behind a firewall or not. Consider the scenario il-

Proceedings of the 15th International Workshop on Database and Expert Systems Applications (DEXA'04)
1529-4188/04 $ 20.00 IEEE

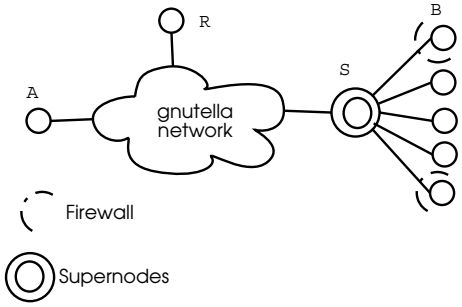

Figure 2. Gnutella **Network Topology with** *repeaters*

lustrated in Figure 2, where A is the poll requestor and B is the poller.

Case 1: A is not behind a firewall and B is behind a firewall (see Figure 1(c)).
To create the PushVote request the Gnutella Push message is modified and sent through the Gnutella network to reach B and to start a Push connection from B to A. The PushVote request is sent along the same path followed by the PollReply. The PushVote request must let B know that someone with the specified *serventID* wants B to start a PushVerification and reply to the TrueVote message. The PushVote request contains the *serventID*, IP address and port of A, a special identifier with the meaning "connect to me" and the encrypted TrueVote request (i.e., encrypted with the public key of the poller, which it sent in the PollReply). The poller B retrieves the TrueVote from the PushVote request, decrypts it and builds the TrueVoteReply and connects to the IP address and port specified in the message. After B sets up a TCP Connection to A, exchange of headers takes place to confirm if everything is ready for the verification. The poller B shows an Identifier confirming about his identity *connection key*. The *connection key* has been chosen to be a random value *rnd* together with the pair (serventID(A), serventID(B)). The Identifier used is unique for each vote verification.

Case 2: A and B are both behind a firewall (see Figure 1(d)).
This mechanism requires a reachable node in the middle of the network for managing the connections needed for the push verification. We call this node *repeater* because its work is to repeat the message from the poller to the poll requestor. Essentially, the role of the *repeater* is to correctly retrieve the TrueVoteReply from the poller and pass it to the poll requestor with the public IP address and port of the poller. More precisely, a repeater session includes the following steps:

1. *Repeater* accepts a connection from A giving a confirmation about its availability
2. *Repeater* retrieves the array (rnd,(serventID(A)) (serventID(B))). This is the identifier of the *repeater* session.
3. *Repeater* saves the attributes describing the *repeater* session in a connection list, to verify if the incoming check connection was the one it was waiting for.
4. *Repeater* sets a timeout for poller connection B, i.e., the maximum time to wait for the poller B to connect.
5. *Repeater* retrieves the *connection key* on connecting with the poller B.
6. *Repeater* controls that the *connection key* passed is the one the poll requestor A asked for; if not, it closes the connection and waits for another connection, until the timeout.
7. *Repeater* retrieves the *repeater* session, saves IP address and port number of the poller B.
8. *Repeater* reads the TrueVoteReply from B and sends it to A.
9. *Repeater* passes the previously saved IP address and the port to A.

When the data is correctly passed to A, the *repeater* removes the connection identifier from the connection list. A repeater session may return a positive response, a negative response, or a connecting error.

3.1. Selection of *repeaters*

A node in the network which wants to become a *repeater* should satisfy the following requirements:

1. It should not be protected by any firewall (at least not on the port on which it is listening to) or NAT, because it must be reachable by every node on the network.
2. It must have sufficient bandwidth and processing power to manage the many connections deriving from being a *repeater*

The reputation in the internal repository of the repeater candidates can be used in this choice, but a random choice should also be effective, given the redundancy-based protection measures that the protocol envisions.

More than one *repeater* is needed in order to have sufficient redundancy and security. Each *repeater* is independent in its work and there are multiple ways to operate: parallel, serial, or mixed. The idea is to use a classical principle of distributed systems, where N participants are considered and it is required to have an agreement on at least the absolute majority of them, to contrast possibly malicious nodes (in our implementation, we assume $N = 3$).

Proceedings of the 15th International Workshop on Database and Expert Systems Applications (DEXA'04)
1529-4188/04 $ 20.00 IEEE

4. Attacks to the vote verification mechanism

The first innovation we introduced in SupRep, the construction of CumulativePollReply messages, has no impact on the security evaluation of the protocol. As SupRep extends P2PRep it is robust against pseudospoofing, ID-Stealth, and shilling attacks. The introduction of repeaters instead introduces several aspects that need to be evaluated. The result of this analysis is that the vote verification mechanism is robust against attacks from malicious nodes.

4.1. Case 1: *A is not behind a firewall and B is behind a firewall*

Attacks may come only from the nodes that are on the path that the Push message follows from the poll requestor A to the poller B. These nodes also saw the PollRequest/PollReply exchange in addition to the current PushVote request. A malicious node M can have noticed that (1) A has launched a PollRequest; (2) an unknown node (since serventID of the poller is encrypted in the PollReply) replied to the PollRequest from the part of the network where the PollReply came from; and (3) A is trying a PushVerification on the same servent (poller B) that replied to the poll (in the Push message there is the same serventID as in the PollReply). The malicious node M cannot make a selection of PushVote requests based upon the servents involved, since M does not know who the poller B is. Therefore, malicious node M cannot block verification of nodes whose vote M does not want to be verified. The malicious node M can block every PushVote message it sees, but this will only limit the ability of A to verify the votes originating in the portion of the Gnutella network beyond the malicious node M. Then, M could also have blocked the votes as they were flowing back to A and this is unavoidable, because the poll requestor cannot choose the path for the PollRequest message.

If the malicious node M was able to access the content of the PollReply, he would get the serventID of the poller B, and then M would see the PushVote message to retrieve the serventID A. This way, malicious node M would have the connection key and could connect to the poll requestor A, which could not distinguish this from the real connection from poller B. But, PollReply messages are encrypted and we assume that M cannot break the encryption and seriously endanger the protocol.

Thus, the push verification protocol is resistant to the above attack.

4.2. Case 2: *A and B are both behind a firewall*

Compared with Case 1, we have here to consider the additional situation when the *repeater* R itself is malicious. The *repeater* R knows the *serventID*s of A and B and that the poll requestor A needs to verify a vote expressed by the poller B. The *repeater* reads a TrueVoteReply message from B (which is not encrypted) and passes it with the IP address and the port of the poller B. The *repeater* R can disconnect to abort the verification process. The protection against this attack is given by the use of multiple *repeaters* and by the adoption of a careful selection of repeaters by the poll requestor A.

5. Conclusions

We have shown the design and implementation of a SupRep aware servent. The protocol is an extension of the P2PRep protocol that considers the presence of super-peers and firewalls in P2P networks. We have also presented a strategy, based on repeater nodes, overcoming the obstacles that firewalls imposes to the vote verification phase.

Acknowledgments

We thank Gianluca Magni and Ivan Piasini for their contribution to the implementation of the system demonstrating the protocol. This work was supported in part by the European Union within the PRIME Project in the FP6/IST Programme under contract IST-2002-507591 and by the Italian MIUR within the KIWI and MAPS projects.

References

[1] K. Aberer and Z. Despotovic. Managing trust in a peer-2-peer information system. In *Proc. CIKM 2001*, Atlanta, Georgia, November 2001.

[2] S. Bellovin. Security aspects of Napster and Gnutella. In *Proc. USENIX 2001*, Boston, June 2001.

[3] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. Managing and sharing servents' reputations in P2P systems. *IEEE TKDE*, 15(4):840–854, July 2003.

[4] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, P. Samarati, and F. Violante. A reputation-based approach for choosing reliable resources in peer-to-peer networks. In *Proc. 9th ACM CCS*, Washington, DC, USA, Nov. 2002.

[5] Gnutella. http://rfc-gnutella.sourceforge.net/.

[6] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina. The eigentrust algorithm for reputation management in P2P networks. In *Proc. WWW 2003*, 2003.

[7] L. Xiong and L. Liu. Building trust in decentralized peer-to-peer electronic communities. In *Proc. 5th Conf. on Electronic Commerce Research (ICECR-5)*, 2002.

[8] B. Yang and H. Garcia-Molina. Designing a super-peer network. In *Proc. IEEE ICDE*, March 2003.

Proceedings of the 15th International Workshop on Database and Expert Systems Applications (DEXA'04)
1529-4188/04 $ 20.00 IEEE