Secure Score Management for Peer-to-Peer Systems

No Institute Given

Abstract. We propose a secure method to manage digital currency and other types of scores in a P2P network. In this method, called MOTHERS[1], each peer i is assigned one or more *score managers* who mediate all transactions in which peer i is involved. Score managers are assigned to a peer using a Distributed Hash Table, so that peers may not choose their score manager. We show that this system has a very low probability of breach, even in highly adversarial conditions where large collections of malicious peers collaborate in an attempt to subvert the system. We also discuss how to make the system more efficient at the expense of decreased security.

1 Introduction

Many P2P applications, such as reputation systems, incentive schemes, and online P2P markets, require that each peer in the system be assigned a score.

In reputation systems (e.g., [2]), each peer may be assigned a reputation score that marks the trustworthiness of that peer. In incentive schemes for participation in a P2P network (e.g., [1]), each peer may be assigned a participation score that reflects the extent of its contributions to the network. In online P2P markets, peers may charge other peers digital currency for files and query responses, and the digital account values of each peer must be maintained.

In each of these systems, a score is kept for each peer, and peers in the network may decide rewards, punishments, and choice of downloads based on the scores of other peers in the network. In these examples, it is clear that the score of a peer must be handled securely. Specifically, peers must not be able to arbitrarily change their own score, and they must only be able to change the score of other peers in a manner specified by the system (e.g. charging for a service).

In a centralized P2P network like Napster [3], these scores may be stored and managed at a centralized trusted authority. However, in a more distributed system like Kazaa, storing an managing scores is more difficult.

The most straightforward way to manage scores in a distributed system is to have each peer i store its own score. If peer j wants to know the score of peer i, it can ask. The problem to such a score management scheme is that it has no protection against dishonest peer. A malicious peer may misreport its

[1] This acronym doesn't stand for anything.

score in order to gain rewards, avoid punishments, or distribute malicious files throughout the system.

We combat this by implementing two basic ideas. First, the peers must not store and report their own scores. Thus, we have a different peer k in the network store and report the score of peer i. We call peer k the *score manager* (or mother) of peer i. Second, since it is in the interest of malicious peers to misreport the scores of their children, we assign multiple score managers to each peer.

We describe this model in detail in this report.

2 Score Management Scheme

In this section, we first describe a simple but realistic model of the network and interactions between peers. We will then present the MOTHER (need a new name) score management scheme, and provide a brief cost analysis.

2.1 Network and Interaction Model

Each node N_n has a score S_n. S_n may only be changed when N_n receives a service from any node N_x, or when N_n receives a service from some node N_x. We only consider transactions involving two peers: one node requesting the service, and the other providing the service.

For each live node N_n, there is a live node M_n that manages S_n. Clearly, we cannot allow N_n to select its manager M_n, otherwise N_n might select a manager that will collaborate with him. Instead, we need a way to assign managers such that managers are effectively randomly chosen from the set of all possible peers.

We say that two nodes are *strangers* if they do not know each other – i.e., they will not be part of the same malicious collective. Although it is possible for two strangers to form a collective together after they meet, it is not likely, since it would require communication outside of the protocol itself (for example, the users of the peers would have to agree to collaborate over some outside channel, such as phone or email, and then re-program their clients to do the same thing).

We canot choose two nodes with complete certainty that are strangers, but we can select M_n such that it is *likely* they are strangers. For example, we may use a DHT (e.g., CAN) to hash the IP address of N_n and follow the routing protocol in order to determine M_n. Because hash functions are difficult to reverse, and particularly because M_n may change over time as the network changes (due to the routing protocol of DHTs), it will be difficult for two malicious nodes to manipulate system such that one is the score manager of the other. Of course, the DHT may work such that one malicious node just happens to be the manager of another malicious node, but we do not see a good way to prevent this problem in any reasonable manner.

Hence, let us assume there exists a function h such that $h(N_n)$ always returns another live node in the system, and with high likelihood, $\mathit{Strangers}(N_n, h(N_n))$. We then assign $M_n = h(N_n)$.

2.2 MOTHERS Protocol

In any transaction between two nodes in the MOTHERS protocol, there are at most four nodes involved:

- N_Q – the peer querying or asking for a service
- N_R – the peer responding or providing the service
- M_Q – the score manager of N_Q
- M_R – the score manager of N_R

Also, we know $\mathit{Strangers}(M_Q, N_Q)$ and $\mathit{Strangers}(M_R, N_R)$ with high likelihood.

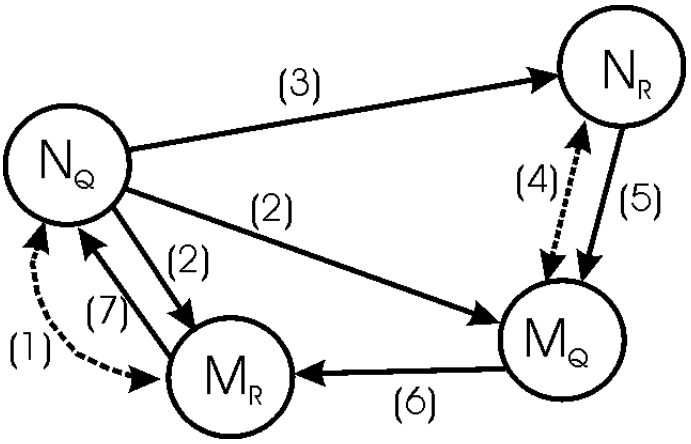

Fig. 1. Illustration of MOTHERS protocol

The protocol of message exchange follows the following steps (as illustrated in Figure 1):

1. N_Q queries M_R for S_R.
2. N_Q registers request with M_Q and M_R.
3. N_Q sends request to N_R.
4. N_R queries M_Q for S_Q.
5. N_R responds, but sends response (or proof of service) to M_Q.
6. M_Q alters S_Q, and forwards response (or proof of service) to M_R.
7. M_R alters S_R, and forwards response (or proof of service) to N_Q.

Let us illustrate the protocol with an example. Consider a web-service discovery system, in which peers make micropayments to other peers for processing queries. In this network, the "score" of a node is its balance of points. Let us look at a single transaction, in which N_Q has a query Q, and wishes for N_R to process the query. Let us assume N_Q only wishes to query nodes with high scores, because high scores imply active, trustworthy nodes. N_Q first queries M_R for S_R.

If S_R is sufficiently high, N_Q sends a message to each of M_Q and M_R, registering query Q with both. N_Q then sends the query to N_R. Let us assume N_R does not wish to provide services to nodes that cannot pay; hence, it first queries M_Q for S_Q. If satisfied, N_R searches its local databases for any service matches. If no matches are found, then it does nothing – no payment is exchanged. If a match X is found, then N_R sends the result to M_Q. M_Q verifies that X matches query Q, and tries to decrement S_Q. If S_Q is already zero, meaning N_Q cannot pay for the service, then M_Q stops processing – again, no payment is exchanged. Otherwise, M_Q sends X to M_R, who likewise verifies that X matches query Q, and increments S_R. Finally, M_R sends X back to N_Q.

In Section 3, we show that every step in the MOTHERS protocol is necessary for security.

2.3 Cost Analysis

A brief cost analysis shows that the MOTHERS protocol adds potentially significant overhead to every transaction. For this analysis, let us define C_h as the cost of executing h, the function to determine a node's manager (e.g., for Chord, $C_h = O(\log(n))$).

First, the number of messages has increased from two to six. Furthermore, depending on the size of the response (in our example, size of X), two of the additional messages may be quite large in size. Finally, and perhaps most importantly, function h must be applied two times: once for N_Q to determine M_R, and once for N_R to determine M_Q (we assume a node "caches" the value of its score manager, so it doesn't need to apply h to itself every time). With DHTs such as Chord, where $C_h = O(\log(n))$ (where n is the number of nodes in the network), then the cost of a transaction has increased from $O(1)$ (albeit with a possibly large constant) to $O(\log(n))$. This scales linearly with the number of mothers assigned to each peer. In addition, because the steps are completely ordered, the response time of a transaction increases as well.

In the next two sections, we address this issue of efficiency. First, we show in Section 3 that every step in the MOTHERS protocol is necessary for security. As a result, any gain in efficiency (whether it be bandwidth or response time) by removing steps will result in less security. Second, in Section 4 we discuss variations of the protocol that do make this tradeoff in a reasonable manner, and discuss when such tradeoffs might be appropriate.

3 Minimality of MOTHERS

We wish to show that the MOTHERS scheme is *minimal*, in that every step is necessary. We will enumerate the steps and illustrate the type of attacks that could subvert the system if the step is not taken, as well as the types of applications in which certain steps are not necessary. In particular, Figure 2 shows the protocol if the application is such that N_Q does not need to pay for service, and Figure 3 shows the protocol if the application is such that N_R does

not need to be paid for service. In our discussions, we will use the example web-services discovery system from before as our example application.

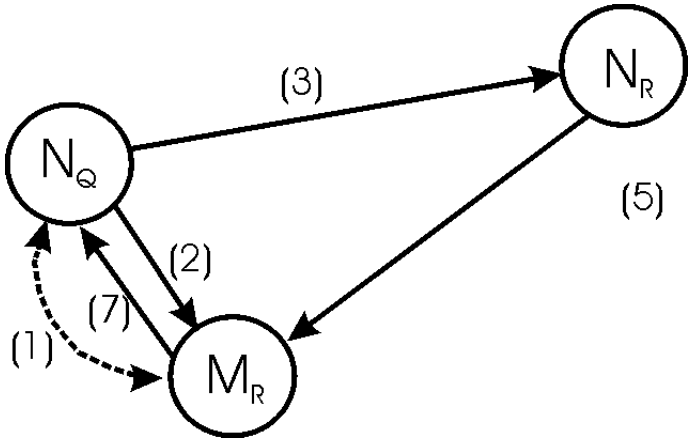

Fig. 2. Illustration of MOTHERS protocol when N_Q need not pay

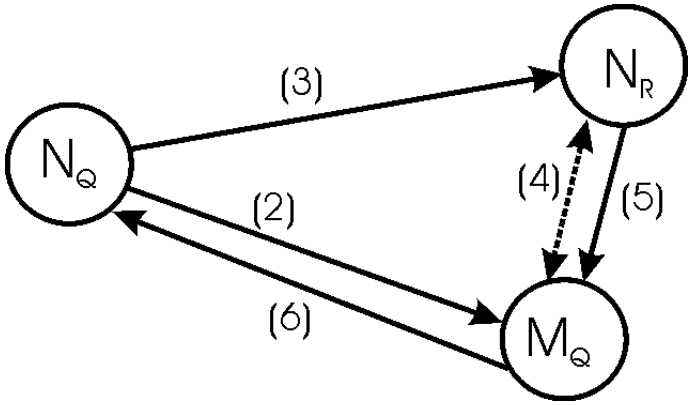

Fig. 3. Illustration of MOTHERS protocol when N_R need not pay

Step 1: This step is clearly necessary on an application-by-application basis. If the application is such that N_Q only wishes to pay for services from N_R if S_R is high, then it needs to query M_R. Otherwise, if the application is such that results from all nodes are equally desirable, this step is not necessary.

Step 2: If queries are not registered, then a node N_a can attack a node N_b (and boost its own score) by generating responses to some imaginary query, and sending these responses to M_b. M_b will decrease S_b and forward the responses to M_a, who will then increase S_a.

Note that if the application is such that N_Q does not need to pay for the service, then N_Q need not register the query with M_Q. Likewise, if the application is such that N_R need not be paid for the service, then N_Q need not register the query with M_R.

Note that the purpose of this step is to prove to M_R and M_Q that N_Q did request a particular service from N_R. There are other ways for N_R to make this proof, other than requiring N_Q to register the request with two additional messages. We discuss alternatives in Section 4.

Step 3: Clearly step 3 is necessary for any meaningful transaction to occur.

Step 4: If S_Q is not high enough to pay for the service, it is in N_R's interest to refuse service. Otherwise, N_R will providing a service that it will not be paid for. N_Q can then mount a DOS attack on N_R by generating many requests it can not pay for.

Note that this step is also necessary on an application basis. If the application is such that N_Q does not need to pay for service, then N_R need not check S_Q.

Step 5: Let us assume the protocol allows N_R to send the response directly to N_Q, and then asynchronously notify M_R and/or M_Q that it has responded. Asynchronous notification is desirable because it reduces the number of messages in the critical path of the transaction, thereby reducing response time. If N_R is malicious, it can simply NOT respond to N_Q (because it requires work to generate a response), but pretend to M_R/M_Q that it DID respond. Hence, it will get paid without rendering the service.

Note that if the application is such that N_Q does not need to pay for this service, then N_R may bypass M_Q safely, and send the response directly to M_R.

Step 6: Let us assume the protocol allows M_Q to send the response directly to N_Q, and then notify M_R asynchronously that S_R should be increased. If M_Q wishes to attack N_R, it may simply NOT notify M_R. (However, this attack is not AS serious, because it requires a mother to be malicious, rather than N_Q and/or N_R. Later, we discuss how to deal with malicious mothers).

Note that if N_R does not need to be paid, M_Q can safely bypass M_R and send the response directly to N_Q.

Also note that M_Q *must* receive the response before M_R does. That is, S_Q must decrease before S_R is increased. Otherwise, if N_R and N_Q are in a malicious collective, they can unfairly boost S_R. Let us assume the protocol is such that N_R sends the response first to M_R, who then forwards to M_Q (rather than M_Q first, then M_R). In this case, N_Q can keep ordering services from N_R even after S_Q is zero. N_R will then send the response to M_R, who will add to S_R. M_R will then send the response to M_Q, who will drop the response since it knows

N_Q cannot pay. However, S_R will already have been increased. Note that this problem is not extremely serious, since N_R will have to do a lot of work to make S_R high. However, it is still an unfair way to generate "false" business.

Step 7: If steps 1 through 6 are in place, then step 7 must occur for N_Q to receive the response.

4 Variations

As we discussed earlier, the MOTHERS protocol does add potentially significant overhead to each transaction. In this section we present several optimizations and variations over the MOTHERS protocol that affect the tradeoff between efficiency and security.

4.1 Optimizations

Here we present two optimizations that allow us to improve efficiency of the protocol without sacrificing security.

First, in step 2, rather than explicitly registering a query, the requester (N_Q in our example) may sign a receipt (Q, N_R) with its private key such that N_R has proof that N_Q asked N_R for the service Q. When N_R sends the response to M_R, it must include this proof. Likewise, M_R must forward the proof to M_Q. This optimization trades off bandwidth consumption with computational load. Depending on the availability of network resources versus computational resources, this option may present a good performance tradeoff.

As a second optimization, rather than having N_Q compute $M_R = h(N_R)$ and N_R computing $M_Q = h(N_Q)$, we can allow each node to report its own score manager (we assume N_i knows the identity of its manager, for all i). Observe that while N_R and M_R must be strangers, N_R has no incentive to falsely report M_R to N_Q. If it does, then it will not be paid for its service. Hence, we can skip the step of N_Q computing M_R, which costs C_h.

The danger of this alternative is that if N_Q falsely reports its manager as some node $M_b \neq M_Q$, N_Q can receive services without paying. M_R will forward the response to M_b, who is presumably in a malicious collective with N_Q. M_b will then forward the response to N_Q without docking S_Q.

To prevent N_Q from cheating, we can have N_Q sign a message indicating that its manager is M_b. N_R will then compute $h(N_Q)$ with some probability p. If M_b accepts the response from N_R (or M_R) without reporting an error, and $M_b \neq h(N_Q)$, then N_R can prove that N_Q is lying. N_R will then report the lie to N_Q's true manager M_Q, who will then punish N_Q by reducing S_Q. Therefore, a node can only cheat for a limited time before it is caught with high probability, at which time all its unfair profits can be revoked. By setting the probability p to $O(\frac{1}{\log(n)})$, the expected cost of a transaction is $O(1)$, rather than $O(\log(n))$ as before.

4.2 Multiple Managers

One problem with MOTHERS is the possibility of a malicious manager. Here we discuss assigning multiple managers to counter this problem.

Let us model malicious behavior in score managers as always returning 0 for other nodes' scores. For example, M_Q may always report $S_Q = 0$, thus preventing N_Q from receiving any services, no matter how much work it has done. We propose a solution where the system has t "hash" functions $h_1, h_2, ..., h_t$, and each node N_a has t managers: $M_{a_1}, M_{a_2}, ..., M_{a_t}$, where $M_{a_i} = h_i(N_a)$. We represent the set of all managers for N_a as $M_a = \{M_{a_1}, M_{a_2}, ..., M_{a_t}\}$.

When a node queries for N_a's score, it will send the query to every node in M_a. It will then use majority vote to determine the actual score of N_a. As long as a majority of the managers are good, then the result of the vote will be good. Hence, we say that M_a is good if the majority of the managers in M_a are good. In terms of the MOTHERS protocol, when a message is sent to M_a, it is sent to every node in M_a, and each manager individually updates its copy of S_a.

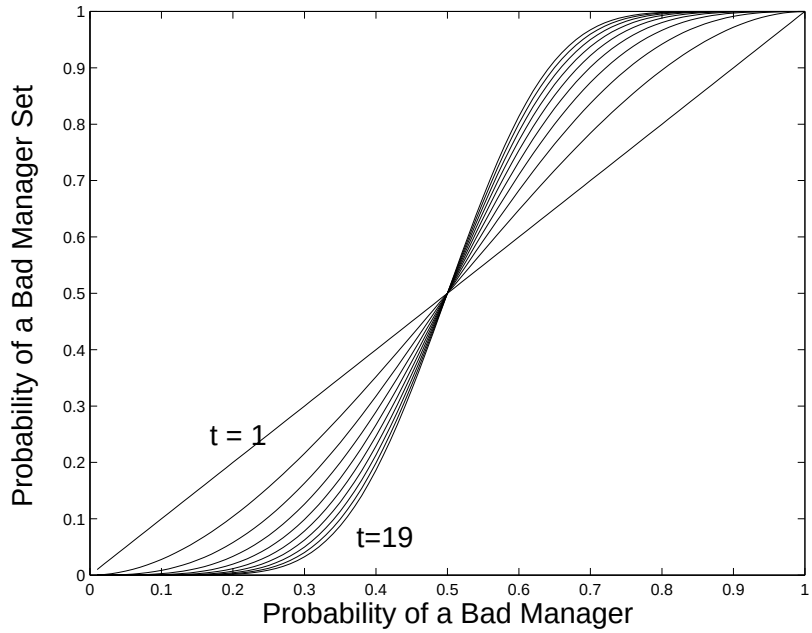

Fig. 4. Probability of a Bad Score

When using multiple managers, selecting a good value of t is presents an important tradeoff between cost and security. Clearly, a larger t results in greater cost. In most cases, it is also the case that larger t results in a higher likelihood of good M_a. In cases of very high probability of malice, however, lower t is better. Figure 4 shows the probability of M_a being bad for some random node N_a, along

the y-axis. Along the x-axis, we vary the probability that a given node is bad, and different curves represent different number of managers. We see that as long as the probability of a random node being bad is less than 50%, larger t results in better security.

Note that it is possible for $h_i(N_a) = h_j(N_a), i \neq j$. In this case, for the above analysis we assume that $h_i(N_a)$ independently "chooses" to be malicious towards N_a across the different hash functions. That is, if $h_i(N_a)$ is assigned to be N_a's manager twice, it may choose to report one true score, and one false score.

5 Conclusion

In this paper we present the MOTHERS score management scheme for P2P networks. We show that MOTHERS is minimal, in that every step is necessary to ensure correct behavior, and present a cost analysis of the protocol. While the MOTHERS protocol incurs a non-trivial overhead to transactions, we present several optimizations and variations that improve efficiency while maintaining a good level of security.

References

1. P. Goelle, K. Keyton-Brown, and I. Mironov. Incentives for sharing in peer-to-peer networks. In *Proceedings of ACM Conference on Electronic Commerce*, October 2001.
2. S. Kamvar, M. Schlosser, and H. Garcia-Molina. The EigenTrust Algorithm for Reputation Management in P2P Networks. In *Proc. WWW*, 2003.
3. Napster website. http://www.napster.com.