TREMA: A Tree-based Reputation Management Solution for P2P Systems

Quang Hieu Vu
ETISALAT BT Innovation Center (EBTIC)
Khalifa University, UAE
quang.vu@kustar.ac.ae

***Abstract*—Trust is an important aspect of Peer-to-Peer (P2P) systems, because in such systems, peers are usually anonymous. A popular method for evaluating trust in P2P systems is to use reputation, where the reputation of a peer is determined based on its prior transactions with other peers. Since no peer has easy access to global knowledge in a decentralized system, the main challenge of this reputation-based method is how to collect and distribute reputation scores of peers efficiently. While several solutions have been proposed to address this challenge, most of them rely on a gossiping algorithm, which is costly and communication-intensive. In this paper, we propose TREMA, a tree-based reputation management solution in which we present a trust model between nodes in the tree, and explain how trust is established and maintained between pairs of nodes. We show that, compared to existing solutions, TREMA allows for scalability and efficient algorithms with low overhead. We present two possible implementations of TREMA, and explain how they could be made stable and robust to network dynamism, thus addressing the greatest weakness of a tree structure. We also analyze each implementation for its security against various adversarial scenarios, and suggest further improvements that are possible for general tree-based systems.**

***Keywords*-Peer-to-Peer; Security; Trust Evaluation; Reputation Management; Tree Structure.**

I. Introduction

Over the last decade, Peer-to-Peer (P2P) systems have received more and more attention from both computer users and researchers. They have become the first choice for large-scale distributed systems due to their scalability. Nevertheless, there are still problems that need to be solved before P2P systems can be truly ubiquitous, one of which is security. Since peers are usually anonymous, security is a problem of greatest concern among people using P2P systems. A popular method for evaluating trust in distributed systems as well as in P2P systems is to base on reputation, where the reputation of a peer is summarized from opinions of all peers who have participated in previous transactions with that peer. For examples, users of eBay [1] and Amazon Auctions [2] are provided a separate channel for feedback. After each transaction, both sellers and buyers can rate each other and the score is kept for a later reference. In this way, from the reputation score of a person, others can decide easily if they can trust that person or not (e.g., a high reputation score is an indicator of having had many successful and trustworthy transactions).

The main challenge of the reputation-based method for trust evaluation in P2P systems is how to collect opinions of all peers in the system about a particular peer, and to provide access to the reputation score to all who request it. In existing reputation-based systems like eBay and Amazon Auctions, the solution to both challenges is to use servers. However, this solution suffers from problems of server-based systems such as network bottlenecks and a single point of failure. An alternative solution is to employ a gossiping algorithm [3], [4], [5], [6] for exchanging knowledge among peers in the system. In this way, after a sufficient number of knowledge exchange steps, every peer should have a global knowledge about reputations of all peers in the system.

The gossiping algorithm can be implemented in two ways. In the first way, each peer itself has to maintain global state and knowledge of the whole system. After each transaction or after some interval time, peers report the score of their partners in new transactions to all other peers in the system. Based on this report, peers update their global state. This method requires that peers keep and maintain reputation scores for all peers, which is inefficient. The second way avoids this problem by letting each peer keep track of the reputation of peers that it has been in transactions with previously. Whenever a peer wants to retrieve the reputation of another peer, it can apply the gossiping algorithm to ask for that peer's reputation from its neighbors, the neighbors of its neighbors, and so on. Combining the feedback with its local knowledge, it can determine the trust value of that peer. Even though these two ways are different, they share the same drawback of the gossiping algorithm: both are expensive in terms of computation and communication costs.

Instead of using gossiping, in this paper, we present TREMA, a Tree-based REputation MAnagement solution. In TREMA, we organize nodes at different positions in a tree-based on their reputation, with peers of higher reputation at higher levels. In this tree structure, reputation of a peer is maintained at its parent. A peer always trusts its ancestors while it is answerable for its descendants. When two peers execute a transaction, a trust route is formed between them. If the transaction succeeds, a reward is given to all nodes in the route. On the other hand, if the transaction fails, all nodes in the route are penalized. The main advantage of TREMA is that it does not incur a high cost in reputation management compared to methods that use the gossiping algorithm for reputation distribution. Furthermore, the flexible design of TREMA allows us to develop a complete system for trust management to use in any existing decentralized P2P system. To sum up, our paper makes the following contributions in the area of P2P security:

- We present TREMA, a general solution for trust management in P2P systems based on a tree structure. Besides, we expose a set of APIs that allows P2P applications to work on top of TREMA.
- We show how to augment a tree with extra links to create robustness and to allow nodes to exchange queries without overwhelming the root. These extra links help to eliminate the problems of bottlenecks and single points of failures in the tree structure.
- We present two possible implementations of TREMA. One is an extension of BATON [7], an existing tree structure. The other is HICON, a novel tree structure. We compare these two systems and show how to improve the weaknesses of both systems.
- We conduct an experimental study on the implementation of TREMA in BATON to evaluate the effectiveness and efficiency of our proposed solution.

This paper is an extended version of a previous paper [8]. In the previous paper, we introduced a secure protocol for trust management in P2P systems based on BATON [7], a tree structure. In this paper, we extend the idea to support all tree structures and also provide further design discussion. The rest of the report is organized as follows. In Section II, we introduce related work in the area of trust management in P2P systems. In Section III, we explain the basic design of TREMA in terms of the trust and security models. In Section IV, we discuss some issues of our basic design, and suggest possible solutions to improve it. In Section V, we describe the general APIs that we are proposing. In Section VI, we first use our design to extend an existing tree structure (BATON) to support reputation management. After that, we present our own tree structure design (HICON). We suggest potential applications of TREMA in Section VII. Section VIII describes our experimental study and its results. Finally, in Section IX, we summarize the important contributions of our design and its potential.

II. Related Work

Trust management in P2P systems can be classified into two main categories: credential-based and reputation-based management. Credential-based management systems employ the classical method where a peer trusts another peer after examining the other peer's credentials. If the credentials satisfy the peer's policy, that peer can be trusted in a transaction. Otherwise, the peer would refuse to be in a transaction with the other peer. The weakness of this method is that it has to rely on servers for keeping every peer's credentials, which is not entirely a scalable method. Moreover, since credentials are usually generated once and stored, past transactions of peers, both good and bad, are not considered. As a result, this method is only suitable for specific kinds of systems with fixed credentials, like access control systems. Examples of systems that apply this trust model include X.509 [9], PGP [10], PolicyMaker [11] and its successors, REFEREE [12] and KeyNote [13].

On the other hand, reputation-based management systems rely on reputation to evaluate the trustworthiness of a node. In general, the reputation of a node is computed based on its previous transactions with other nodes in the system and how they rated these transactions. Reputation-based management systems can be further classified into two subcategories. One type of system considers only the reputation of an individual, like those in [3], [4], [5], [6], [14], [15], [16], while the other takes into account social relationships between nodes in addition to individual reputation, such as [17], [18]. Since no nodes know of all nodes in the system, reputation of nodes have to either be collected and stored on servers for reference or distributed to all nodes in the network by the gossiping algorithm. Both of these methods are not viable for large networks because the first method is not scalable while the second method is expensive.

In the field of data structures, the structure of a tree has a very important role. NICE [19] can be used to do scalable application layer multicast [20] by using the idea of overlay trees for efficient content distribution. However, very few networks proposed so far uses the topology of a tree. In this kind of structure, if the standard query processing algorithm is used, nodes near the root will be accessed many times more compared to nodes near the leaves, and hence congestion at the root or nodes near the root may happen. This is not acceptable in P2P systems. To avoid this problem, P-Tree [21] suggests a use of partial tree structure. In this method, each leaf node in the tree is represented by a P2P node while internal nodes are all virtual. Each P2P node maintains a path from the index root to the leaf node. As a result, queries can be processed at any node without pushing all queries to a special node. Note that, however, if a node has to maintain the whole tree structure, the maintenance cost is very expensive and not suitable for P2P systems. Alternatively, BATON [7] creates links between nodes at the same level in the form of routing tables. Consequently, queries can be processed at any node in the tree without going through the root. Nevertheless, these systems focus only on range query processing, and not trust management.

III. Basic Design

A. Trust Model

TREMA consists of peers arranged by their reputation. Peers of higher reputation occupy positions at higher levels in the tree, with each parent having a higher reputation score than its children, and so the root node is the peer with the highest reputation. Peers of higher reputation are accorded higher privileges of some kind, to provide incentive

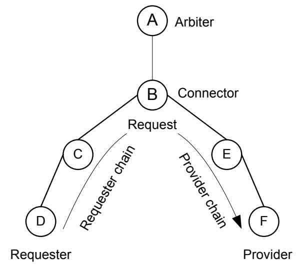


Figure 1. Trust relationships in a trust route

for nodes to increase their own reputation. We develop the following terminology and use it to present the model of trust relationships between nodes in the tree.

- *Trust link*. A link exists between a peer and its child, and this denotes a link of trust. We say that (1) the child peer in this link trusts its parent because the parent has a higher reputation than itself, and (2) the parent is answerable for the child. The latter point means that any misbehaved action on the part of the child reflects poorly on the parent as well, and the parent is also held accountable for any misbehavior of the child. This is desirable because it is every peer's responsibility to minimize the presence of malicious peers entering the network as children. Trust links are inherently transitive, because a child that trusts its parent would also trust its parent's parent of higher reputation, while a parent is accountable for its children and thus its children's children as well.
- *Trust chain*. A chain of trust is formed by consecutive trust links. In such a chain, we say that the lowest peer trusts the highest peer, based on transitivity of trust in our model.
- *Trust route*. A trust route is the path between any two peers in the tree. It is composed of one or two trust chains that meet at a common ancestor of the two nodes. We call that ancestor the *connector* of the route. The trust route also includes the connector's parent, which we label as the *arbiter* of the route. A trust route is formed when a peer requests content from another peer in the system. The former peer is known as the *requester*, and the latter is the *provider*. The trust chain from the requester to the connector is called the requester chain, and that from the provider is called the provider chain. Figure 1 illustrates the relationships mentioned here.
- *Transaction*. A transaction is initiated by a requester, by sending a request through the tree to a chosen provider. The provider responds with the appropriate content to the requester. Transactions occur over a trust route in our tree, and they have a *transaction outcome* in the form of a report sent out by the requester. A positive outcome indicates a successful transaction when the requester is satisfied with the received information. Conversely, a negative outcome indicates a failed transaction when the requester is not satisfied with some part of the received information.
- *Rewards and punishments*. To give a reward means a peer increases the reputation score of a child peer, and a punishment is the converse, a decrease in the reputation score of the child peer. Rewards and punishments are managed based on the transaction outcomes reported by requesters.

B. Trust Management

This subsection describes how trust in our model can be managed. There are two possible outcomes of transactions each of which is dealt with in a separate way.

- *Successful transactions*: if a successful transaction occurs between two nodes via a trust route, parent nodes would reward the child nodes. The rationale is that rewarding a child would allow it to be trusted by more nodes, and hence to increase its potential for bringing in more transactions for itself. This would lead to more opportunities for the parent node to earn its own rewards. In general, after a successful transaction, the arbiter rewards the connector, the connector rewards both the children in the requester and the provider chains, and so on, downward both trust chains. The only exception is the requester, which does not get any reward for initiating a request, since it adds no value to the network.
- *Failed transactions*: for a failed transaction, the converse happens. The arbiter punishes the connector, which in turn pushes the blame downward the tree from parents to children in both chains. The requester again is unaffected by the punishments because it has nothing to gain or lose for accurately reporting the outcome of the transaction. A truthful report would, however, increase the effectiveness of the whole network. To prevent the malicious scenario of a node deliberately reporting multiple failed transactions, a parent might keep track of node failure reports, and identify any nodes that are misbehaving in this way. The parent could then terminate trust links with any evil node, deeming it to be deliberately causing trouble by either requesting content from reputably bad nodes, or inaccurately reporting many failed transactions.

This design leads to two main implications. On the one hand, nodes will try to maximize the number of successful transactions and minimize the number of failed ones, in order to optimally increase their reputation. This selfish and self-centered behavior, however, allows for optimal gains for the system as a whole, because each node selfishly seeks to maximize its own rewards and to do so, it has to shrewdly monitor its children and their behavior in transactions. On the other hand, a node would quickly break off links with children that result in many failed transactions and refuse to forward transactions from such nodes, because it is being

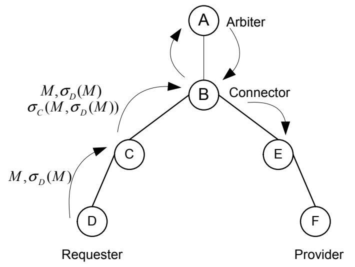


Figure 2. Message forwarding over a trust route. For brevity, we only show the first two of a chain of signed messages.

held accountable for the behavior of its children. At the same time, a node would be willing to forward requests and content from reputable nodes or new nodes because doing so would give it the potential to increase its reputation.

C. Security Management

We claim that two nodes in TREMA can exchange messages over a trust route in an accountable and authenticated manner as follows. When the requester sends a request, the request is forwarded via the requester chain of trust. Forwarding a message means signing the message and sending it over a trust link. Here, a child node always trust its parent to forward the request properly, and a parent node is willing to be accountable for its children because it stands to gain in reputation if the transaction is successful. Note that if the transaction fails, the parent could always terminate the link to its children.

In case of the arbiter, when it receives a message that is signed by one of its children, which is the connector of the trust route. The arbiter signs that message, indicating it has seen the request and is willing to trust that the connector child will do a proper job forwarding that request. The signed request is then sent back down to the connector, who continues forwarding it down the provider chain to the provider. At the provider chain, each node is willing to forward the message down the tree because the message has been signed by its parent, whom it trusts. The provider upon receiving the request can obtain the entire trust route by observing the signatures on the request. When the provider responds with content, it sends both the original multiply-signed request and the content signed by itself. This message is forwarded back via the same trust route, and the reverse happens, with each node signing the message and forwarding it, including the arbiter again. The incentive for forwarding remains the same: a node could stand to gain from aiding a successful transaction, and is able to exert control on any misbehaving children.

Finally, once the requester receives the requested content, it first evaluates the transaction and assigns the transaction either a positive or negative score, and then sends this score in an encrypted message to the arbiter, via a direct connection. Based on the received score, the arbiter will give the appropriate reward or punishment for the transaction.

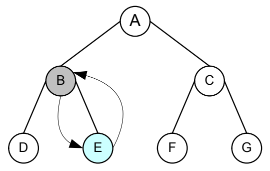


Figure 3. Swapping positions between nodes

Figure 2 shows an example of message forwarding over a trust route. Note that requests and complaints are also communicated securely and authentically by encrypting then signing the messages. This design ensures that only the correct nodes would be able to read the messages.

D. Node Ranking Management

If we want to know reputation score of a node, we have to ask its parent, since the parent in our tree is of higher reputation and is thus more trustworthy. If an internal node cannot accomplish its task or turns malicious, we should replace it with a better node. Since a node may never want to step down from its position, we have to exert control over that node through its parent.

Additionally, reputation scores of a node is not only stored at the parent but also at the grandparent. Consider the situation where a node now has a reputation lower than that of its child, implying that the tree is currently not well-formed. The solution to this situation is to swap the positions of these two nodes through a swap operation, and that can only be done from the position of the parent of the ill-placed node. By changing positions, these nodes also exchange knowledge information of their children and reputation of these children they are keeping.

An example of node swapping is shown in Figure 3 in which node B has to swap its position with its child E because E has a higher reputation. Actually, since A knows reputation of all B, C, D, E, F, G, it can also swap positions between B and G if G has a better reputation than both B and E. This sort of swapping can be done if A wants to assign a node that is known to be trustworthy from another subtree to be the parent of a subtree that could possibly contain colluding malicious nodes.

E. Concerns of a Tree Structure

Tree structures are widely used in computer systems. They are especially good for storing and retrieving data. Among various data structures, the tree is a dominant structure used in the area of databases. However, there are usually some concerns when employing tree structures in distributed systems. Below are the three greatest concerns.

- Bottlenecks and single points of failure could exist at the root or at nodes near the root in a tree. If every query has to reach the root node before it is processed, the root will become overloaded by queries. As a result, this would be a problem in large networks.

- Usually, the cost of query processing (in terms of the number of search steps) is bounded by the height of the tree. Therefore, if the tree is skewed and unbalanced, the cost of searches might be high.
- In a weakly-connected tree structure, failure of a node may partition the tree completely. As a result, in distributed systems, trees are often augmented with extra links to avoid this problem.

Keeping in mind these concerns, we will design a system without such problems in Section VI.

IV. An Improved Model

The above basic model works well under an assumption that the information given by a node to another node about its children is always correct. In other words, all internal nodes can be trusted in giving information. This is because if an internal node is bad, it can return wrong reputation results about its children to other nodes. For example, a malicious node could return a good reputation score about a bad node or a bad reputation score for a good node. To avoid the problem of the basic model, we introduce a new type of score called a *reference score* for internal nodes. The reference score is used to reflect exactness of information a node gives to others. Now, the trust value of a node is based on not only its reputation score but also the reference score of its parent. In other words, if a node always gives correct information about its children to others, we should trust its information. However, if a node often makes mistakes or gives incorrect information deliberately, our trust in information provided by that node is reduced. Similar to reputation score, a reference score of a node is stored at its parent. So now, as illustrated in Figure 4, before each transaction, a node y should find not only x's reputation score, which is stored at z, the parent of x, but also z's reference score, which is stored at t, the parent of z and after each transaction, y updates scores for both x and z.

The problem now is how to evaluate correctness of information received from z to give feedback of a score after a transaction. Here, we propose a simple solution as follows. When a node is asked about reputation of its children, in addition to giving the total reputation score, it also gives the standard variation of the scores calculated from previous transactions. As a result, the correctness of received information is evaluated by both the reputation score and the standard variation. For example, if the result of the transaction falls far away outside the standard variation, the node giving information should be rated with a bad reference score.

That is not all. Assume that in the worst case, x, z, and t are all malicious peers and they cooperate with each other. If t gives a wrong reference score for z while z gives a wrong reputation score for x, y would still be cheated. To further enhance security, y can also ask reference score of t from its parent. In general y asks for reference scores of a chain

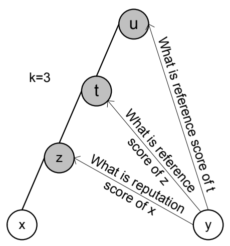

k=3
u
t
z
x
y
What is reference score of t
What is reference score of z
What is reputation score of x

Figure 4. A k=3 reference chain

of k ancestors of x in which k is a configurable parameter of the system. Note that since these k nodes form a chain, the cost of lookup algorithm and update algorithm is just $logN + k$. By setting k with a large number, the system becomes strong against collaborative malicious peers. A worry is that k may have to be large, and hence it may be costly. However, since nodes in the system cannot determine the location of them in the tree structure, they have to follow the join algorithm, which scatters nodes along the system to make the tree balanced. As a result, forming a long chain of malicious peers connected by parent-child links is not easy. An example of a $k = 3$ reference chain is shown in Figure 4 in which y asks z for reputation score of x, t for reference score of z and u for reference score of t.

Another technique which can be used by a group of malicious nodes to trick other nodes is to create fake transactions and report good results to their parent to increase their reputation score. To avoid this problem, we just use a simple technique in score calculation as follows. First, we do not simply consider the number of successful transactions as the score. Instead, we limit the score at a maximum value, and the score of a node can only reach that maximum score. Second, we calculate not only the number of successful transactions but also the number of *different* successful transactions of nodes. By "*different*", we mean that transactions of the node that are done with different nodes. As a result, even though a node may have many good transactions with a specific node, it still has a low score if it has many other bad transactions with other nodes.

V. APIs

Based on the design of TREMA as described in the previous sections, the general APIs we need to provide can be divided into three categories. The first category contains APIs for trust management, which perform the behaviors discussed in our earlier sections. The second category is a set of indirection APIs, which are function calls that applications might need to call. These APIs also provide indirections to the lower P2P network layer. This design allows us to keep the underlying layer completely hidden from the applications that use our framework. Furthermore, with this separation, it is less likely that applications would make incorrect calls that have not been secured by the framework. The third set includes APIs of the underlying P2P network that are augmented with additional functionality to support TREMA. We respectively discuss APIs in these three categories in the rest of this section.

A. Trust Management APIs

- *Reputation-Query*: this API takes in a target node and returns the reputation of that node by querying its parent over a trust route. This reputation request is considered as a transaction and will have an accompanying transaction outcome report from the requester.
- *Reputation-Update*: this API is called by a node that wishes to update the reputation of its child. The update is done by sending the request up to the k reference chain, and upon getting the approval proceeding to change the reputation of the child.
- *Reputation-Complaint*: this API is called by a node to lodge an official complaint up to the k reference chain when the node feels that its reputation has been unfairly modified.
- *Transaction-Report*: this API reports the outcome of a particular transaction to the arbiter of the trust route. Given the report, the arbiter decides whether to direct a cascading series of rewards or punishments.

B. Indirection APIs

- *Node-Find*: in general, when a new node joins the system, it needs to perform bootstrapping. In most decentralized P2P systems, this action involves finding and connecting the node to an existing node in the system. While this function is usually provided at the P2P network layer, we use an indirection API to securely expose it to upper layer applications.

C. Augmented APIs

- *Node-Join*: this API is called by a node that wishes to join the network. The API allows the new node to find its position in the tree structure, and is only triggered after calling *Node-Find* described above. Basically, upon receiving the contact node from *Node-Find*, the new node sends a "Join" message to the contact node. If that node is the correct parent, the new node is accepted as its child. Otherwise, the contact node forwards the request to either its parent or a child that is more suitable. In this manner, the request can be forwarded through the tree until the correct parent of the new node is found within $O(\log N)$ steps. Note that for correct forwarding, the tree needs to have positional determinism, where the position of a node can be determined given the state of the network. Both our proposed tree implementations support this.
- *Node-Depart*. this API is called by a node when it wishes to leave the network. This allows the system to establish new tree links and close down old ones where applicable. In a best-case scenario without failures, calling this API allows efficient updating of any routing table and link, and minimizes disruption to the network due to nodes leaving the system.
- *Node-Failure-Discovered*: this API is called by a node that discovers one of its neighbor nodes is not responding, presumably because that neighbor has failed. Calling this API would set off the appropriate measures to confirm that failure and establish new links as if that failed node had called *Node-Depart*.

VI. System Design

At this point, with the APIs firmly laid out, we are able to describe in greater detail how to extend TREMA to use in two tree implementations: BATON, an existing tree-based framework and HICON, a novel scheme we propose in this paper. In essence, we try to place TREMA on top of existing networking frameworks that provide the topology of a tree. The challenge here is to ensure that we can effectively and efficiently implement our proposed trust management APIs above, and also use the desirable properties of these frameworks to address the weaknesses of the basic tree.

A. BATON

In this section, we first describe the structure of BATON. After that, we introduce the way to deploy TREMA on it.

1) Basic Structure

In BATON, each peer participating in the network is responsible for a node in the tree structure. The position of a node in the tree is determined by a pair of a *level* and a *number*. The level specifies the distance from the node to the root while the number specifies the position of the node within the level. BATON uses three kinds of links to make connections between nodes: parent-child links are used to connect children and parents; adjacent links are used to connect adjacent nodes; and neighbor links are used to connect neighbor nodes at the same level having a distance 2^i from each other. Neighbor links are kept in two special sideways routing tables: left routing table and right routing table. An example of a BATON tree is shown in Figure 5. Note that in this figure, only neighbor links of the grey node are shown.

BATON controls the balance of the tree by forcing that if a node has a child, it has to have a maximum number of possible neighbor links within its level, or in other words, have full routing tables. As a result, when a node receives a join request from a new node, it can only accept the new node as its child if it has full routing tables. Otherwise, depending on the condition, the request is forwarded to either its parent, its neighbor or its adjacent node. In case of node departure, if a node is a leaf node and none of its neighbors has children, it can leave the network. Otherwise, if it is either an internal node or a leaf node, whose neighbors have children, it has to find a replacement node, which is

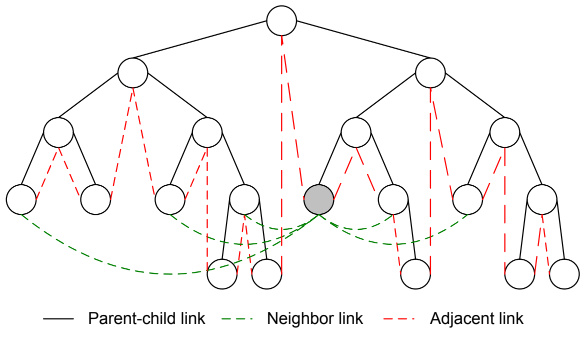


Figure 5. BATON structure

a node in the previous case to replace its position in the tree structure. In particular, by employing sideways links, BATON does not suffer any of the tree issues discussed in Section III-E.

2) TREMA Deployment

Since the most important issues in deploying TREMA are how reputation of a node is looked up and how transaction results are reported to responsible nodes, we focus our discussion of these issues.

- *Reputation lookup*: before each transaction, nodes exchange information about their location in the tree to each other. Knowing the location of a node x, its partner y can infer the location of x's parent, which is z as below:

$$z_{level} = x_{level} - 1$$

$$z_{num} = \begin{cases} x_{num}/2 & \text{if } x_{num} \text{ is even} \\ (x_{num}+1)/2 & \text{if } x_{num} \text{ is odd} \end{cases}$$

  Note that in the tree structure, the level is setup increasingly from the root to the leaf starting at 0 while the number is assigned from the left to the right of each level starting at 1. Now, knowing the location of z, y can issue a query to lookup x's reputation towards z. The algorithm of sending a query towards a node knowing its location is represented as in Algorithm 1. Since at each step, this algorithm makes the search space reduce by half, it is guaranteed that after maximum $O(logN)$ steps, the query should reach the destination node z. When z receives the query, it returns the reputation score of x to y. Note that if x does not tell a truth about its location, and hence when y issues the query either z can not be found or z is not the parent of x. As a result, x can be considered as a bad node.
- *Transaction result report*: after each transaction, a similar process is done to report the result of the transaction between partners to their parent. In particular, each peer rates the transaction by giving its partner a score in a range of [-1.0, 1.0]. Depending on the level of satisfaction or dissatisfaction, a value is given in which a positive score is used to indicate a good transaction while a negative score indicates a bad transaction.

Algorithm 1 :Query (level l, number n, node z)

```
$l_{node}$ = level of the current node
$n_{node}$ = number of the current node
**if** $l_{node} = l$ **then**
    $t$ = the nearest node to $z$
    $t$.Query($l$, $n$, $z$)
**else**
    **if** $l_{node} > l$ **then**
        $t$ = a child of the current node
        $t$.Query($l$, $n$, $z$)
    **else** $\{l_{node} < l\}$
        $t$ = parent of the current node
        $t$.Query($l$, $n$, $z$)
    **end if**
**end if**
```

B. HICON

In this section, we first introduce the basic design of HICON. After that, we show how to deploy TREMA on HICON.

1) Basic Structure

HICON stands for Hierarchical IP Clustering Overlay Network. Nodes in a HICON tree are arranged such that we have the following condition hold for every pair of nodes x and y in the network:

- x is an ancestor of y if and only if, for their common IP prefix P, x has the highest reputation among all nodes with the same IP prefix P.

This condition ensures that for a given state of all nodes in the network, there is a fixed and logical tree structure based on the IP addresses and the reputation scores of nodes, and each node will have its own fixed position in the tree. We call this hierarchical IP clustering because we can build this tree by first clustering all nodes into groups with the same prefix, and then appointing a parent node with the highest reputation as the leader in each cluster. This process can be repeated for clusters of other prefix lengths until we get a complete tree.

By varying the lengths of the prefixes considered, we can change the expected height of the tree and the maximum number of children a node might have. As an example, Figure 6 shows nodes in a HICON tree where we have considered prefixes of 1, 2, 3 and 4 bytes, i.e., 8-bit increments. We see that the node A with IP address 18.4.6.5 has reputation 1000 and is the ancestor of all nodes with IP prefix "18.*". Node B is a child under node A but is the parent of the cluster of nodes with IP prefix "18.4.*", which includes nodes E and F.

In general, in IPv4, if we consider prefixes of B-bit increments, the expected height of the tree will be $32/B$

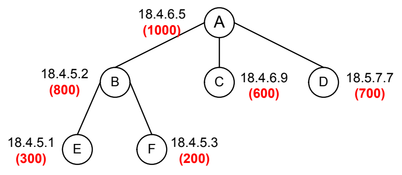


Figure 6. Nodes in a HICON tree

and the maximum fan-out would be 2^{B+1}. This is useful because the height and fan-out of this tree is configurable based on the parameter B. In IPv4, having $B = 8$ gives a very short tree of height 4, but in IPv6, this would give a reasonable height of 16.

The basic structure in HICON gives us the properties of scalability, and some efficiency that a tree inherently has. However, in order to achieve robustness to network dynamism, we introduce the idea of successor nodes and links. In the improved design, in addition to links a node x has to its parent and children, x has a *successor link* to the child node that has the highest reputation among its children. This child y is the *successor node* of x. When x departs or fails, y will take over the position of x. Besides, *successor links* are also maintained among y and its siblings (other children of x), and between y and its potential parent z, which is the parent of x (z will be the parent of y if y comes to replace the position of x). Note that within the trust model, all these successor links are also trust links over which communication can happen, and are recognized and acknowledged by nodes on both sides of the links.

3) TREMA Deployment

For TREMA to work with HICON, we need to implement the augmented network APIs in the following way.

- *Node-Join*: to implement the *Node-Join* procedure, we first ignore the reputation score of x and determine the position of a new node x simply based on its IP address. In particular, when a node y receives a join request from x, if x has some common IP prefix P with y and P is within the cluster that y is the leader of, x is a descendant of y. In this case, y searches its children to find the one whose IP has the longest matching prefix with the IP of x, and forwards the join request of x to that node. If no such child exists, y accepts x as its child. In the other case, if x does not have common IP prefix with y or x does not belong to the cluster of y, y forwards the join request of x to its parent node, who has a broader view, to process the request. Finally, once x is accepted as a child of a node z in the tree, the reputation score of x is used to compared to the scores of other children of z so that x is put it at the correct position.
- *Node-Depart*: when a node x is leaving, x informs its neighbor nodes of its intended action, so that they can update their link information. The successor child of x at this point proceeds to take over the position of x, and uses successor links to establish connections with its new parent and children. Links from x are then removed from the network.
- *Node-Failure-Discovered*: when the failure of a node x is discovered, the discovering node confirms this with the parent of x, who can then proceed to find the successor child of x to take over the position of x. This reduces to the situation of a node departure, with the successor child taking over the position of x.

Additionally, we need to implement these methods for trust management.

- *Reputation-Query*: given a target node x, the requester formulates its query and forwards the query to the parent of x. This is done in a method similar to *Node-Join* where we try to find the position of a node. The same procedure is used here to determine who is the parent of x and how to forward the query to that node. In particular, when a node y receives a query for the reputation of a node x, if x is a descendant of y (i.e., x has a common prefix P that is within the cluster of y), the parent of x must be in the subtree of y. In this case, y chooses the appropriate child node to forward the query to. Otherwise, y forwards the query to its parent, who can continue to forward the query to the correct parent of x. Finally, once the query reaches the parent of x, a trust route is established, and that route can be traversed in reverse to get the response back to the requester.
- *Reputation-Update*: there is no special implementation needed for this tree, we just need to follow the general APIs' specification.

C. Comparison of BATON and HICON

In this section, we make a comparison about the two tree structures presented above. In general, there are three main differences between these structures, as described below.

- HICON is based on a multi-way tree structure whose fanout is 32 for prefixes of 4-bit increments, while BATON is based on a binary tree structure whose fanout is fixed at 2. As a result, in terms of reputation lookup boundary, HICON is better.
- In HICON, reputation is looked up through a chain from child to parent to child. As a result, if nodes are in different branches of the tree, the query has a high potential to be forwarded to the root or nodes near the root, and hence bottlenecks as well as single points of failure may still be problems (even though it is not as severe as in centralized servers based systems). On the other hand, BATON employs sideways routing tables in the process of reputation lookup, which can avoid forwarding the request to higher level nodes.
- In HICON, nodes are grouped by their IP address while in BATON, nodes are distributed randomly to guarantee the balance of the tree.

D. *Further Improvements for Tree Structures*

From the above comparison between BATON and HICON, we can see the advantages and disadvantages of each tree structure compared to the other. As a result, in this section, we propose further techniques to improve the two tree structures.

- BATON can employ a higher fanout tree structure as that in HICON. The reason why BATON is based on a binary tree structure is because a higher fanout tree structure makes it more difficult to manage tree nodes to support range queries, which is the main motivation of BATON. In our system, since our target is to support reputation management, it is possible to extend BATON to support a higher fanout tree structure. On the other hand, HICON could leverage the idea of sideways links (routing tables) in BATON to provide sideways travel of reputation messages, and hence it can totally eliminate the problem of bottleneck and single point of failure as BATON does.
- Another technique, which can be used for both systems is to allow a node to have multiple parents. This technique makes the system less susceptible to being partitioned in case of a massive failure.
- An interesting feature of HICON is that nodes are grouped by their IP address. Based on this feature, we can employ a special kind of reputation called social reputation in HICON. As pointed out in [18], [17], the reputation of a node is affected by the reputation of the society it belongs to and vice versa. The design of HICON is very suitable to employ this kind of reputation scheme since all nodes in the same group have the same prefix IP address, and hence they usually come from the same organization or region.

VII. Potential Applications

A. *General P2P Applications and File-sharing Systems*

Currently, file-sharing applications are dominant in P2P applications. All well-known P2P applications such as Gnutella [22], Napster [23], BitTorrent [24] are file-sharing systems. A security challenge in these systems is that when a peer wants to download a file, it issues a query and may receive a lot of answers from other peers. What is a good peer that the peer issuing the query wants to download the file from? How can a peer be guaranteed that it does not download a wrong file or a file containing virus from another peer? A reputation management system like TREMA can be used in this case to answer these two questions. Furthermore, reputation can also be used to evaluate the quality of services. As an example, a node with high reputation is not only a trustee node but also a node which can provide high bandwidth for downloading.

B. *Spam Reporting Systems*

If anyone wants to build a distributed spam reporting system that is similar to Blue Frog [25], developing from TREMA will be a good start. In general, this system will consists of two basic operations: spam reporting and spam lists forwarding. In particular, when a node receives a spam, it will report the spam and the contact information to a subset of nodes in the network. The recipient nodes will then decide for themselves if the given thing is really a spam and if the contact information is matched, given both the message and the reputation of the claimant. If the report is correct, they will add the spam to their spam lists and forward these lists to other nodes, together with the identifier of the one who claimed it. Finally, they will update the reputation of the sender. Usually, these spam lists could be sent around, and some way of compounding reputations will be worked out. In this way, every node has a list of email addresses it thinks spams, and the contact information thereof. It is interesting to note that without a careful management of reputations, this system may devolve into a way to DDoS any website a malicious node wants to.

VIII. Experimental Study

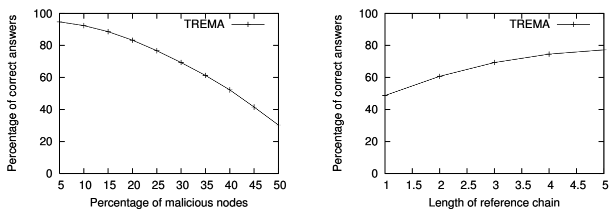
To evaluate the performance of our proposal, we have implemented an extension of BATON [7] to support our security protocol. We tested our system in a network of 1,000 nodes, where exists two kinds of nodes: good nodes and malicious nodes. We just make a simple assumption that that good nodes always do good transactions and give correct answers if they are asked for reputation of their children. On the other hand, malicious nodes always do bad transactions and give incorrect answers about reputation of their children.

A. *Effect of Varying Number of Malicious Nodes*

We first evaluate the effect of varying number of malicious nodes on the strength of the system. The result is displayed in Figure 7(a) in which the x-axis presents the percentage of bad nodes in the system while the y-axis presents the percentage of correct answers about reputation of nodes. The length of reference chain in this experiment is fixed at 3. The result shows that our system can suffer up to 20% of malicious nodes while still provide good answers for a reputation of nodes: more than 80% of answers is correct. It is because in order to fully cheat other nodes, malicious nodes have to form a subtree height greater than 3. However, it is difficult to do that since nodes are distributed equally in the leaf level to keep the tree balance.

B. *Effect of Varying Length of Reference Chain*

In this section, we vary length of reference chain from 1 to 5 while keeping the percentage of malicious nodes at 30%. The result is displayed in Figure 7(b). The result confirms that the system increases its strength with the increasing length of reference chain.



(a) Effect of varying number of malicious nodes (b) Effect of varying length of reference chain

Figure 7. Experimental results

IX. Conclusion

In conclusion, in this paper, we proposed TREMA, a general solution for reputation management in Peer-to-Peer systems based on a tree structure. By using a tree structure, TREMA can avoid the high cost of broadcasting messages that is seen in gossiping-based solutions. At the same time, TREMA does not suffer the problem of bottlenecks and single points of failure as seen in server-based solutions through the employment of extra links in the tree structure. We came up with two specific tree structures to implement TREMA on. One is extended from BATON [7]. The other, HICON, is a novel design. We made a comparison between these tree structures, showing their advantages and disadvantages. From there, we suggested further improvements to both. Finally, we conducted experiments on the implementation of TREMA on BATON to show the effectiveness and efficiency of our proposed solution.

References

[1] eBay, "http://www.ebay.com/ (last accessed: Jan 09, 2012)."

[2] Amazon Auctions, "http://auctions.amazon.com/ (last accessed: Jan 09, 2012)."

[3] S. Kamvar, M. Schlosser, and H. Garcia-Molina, "EigenRep: Reputation Management in P2P Networks," in *Proceedings of the 12th WWW Conference*, 2003, pp. 123–134.

[4] S. Lee, R. Sherwood, and B. Bhattacharjee, "Cooperative peer groups in nice," in *Proceedings of the 22nd INFOCOM Conference*, 2003, pp. 1272–1282.

[5] B. Dragovic, B. Kotsovinos, S. Hand, and P. R. Pietzuch, "Xenotrust: Event-based distributed trust management," in *Proceedings of the 2nd International Workshop on Trust and Privacy in Digital Business*, 2003, pp. 410–414.

[6] L. Xiong and L. Liu, "Peertrust: Supporting reputation-based trust for peer-to-peer electronic communities," *IEEE Transactions on Knowledge and Data Engineering*, no. 7, pp. 843–857, 2004.

[7] H. V. Jagadish, B. C. Ooi, and Q. H. Vu, "Baton: A balanced tree structure for peer-to-peer networks," in *Proceedings of the 31st VLDB Conference*, 2005, pp. 661–672.

[8] Q. H. Vu, "SPP: A Secure Protocol for Peer-to-Peer Systems," in *Proceedings of the 2nd International Conference on Advances in P2P Systems (AP2PS)*, 2010, pp. 1–6.

[9] International Telegraph and Telephone Consultative Committee (CCITT), *The Directory - Authentication Framework, Recommendation X. 509*, 1993 update.

[10] P. Zimmermann, *PGP Users Guide*. MIT Press, 1994.

[11] M. Blaze and J.Feigenbaum, "Decentralized Trust Management," in *IEEE Symposium on Security and Privacy*, 1996, pp. 164–173.

[12] Y.-H. Chu, J. Feigenbaum, B. LaMacchia, P. Resnick, and M. Strauss, "REFEREE: Trust management for Web applications," *Computer Networks and ISDN Systems*, vol. 29, no. 8–13, pp. 953–964, 1997.

[13] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis, *The KeyNote Trust Management System, Version 2*. RFC-2704. IETF, 1999.

[14] K. Aberer and Z. Despotovic, "Managing Trust in a Peer-2-Peer Information System," in *Proceedings of the 9th International Conference on Information and Knowledge Management*, 2001, pp. 310–317.

[15] F. Cornelli, E. Damiani, S. D. C. di Vimercati, S. Paraboschi, and P. Samarati, "Choosing Reputable Servents in a P2P Network," in *Proceedings of the 11th WWW Conference*, 2002, pp. 376–386.

[16] E. Damiani, D. C. di Vimercati, S. Paraboschi, P. Samarati, and F. Violante, "A reputation-based approach for choosing reliable resources in peer-to-peer networks," in *Proceedings of the 2002 ACM Conference on Computer and Communication Security*, 2002, pp. 207–216.

[17] J. Pujol and R. Sanguesa, "Extracting reputation in multi agent systems by means of social network topology," in *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, 2002, pp. 467–474.

[18] J. Sabater and C. Sierra, "REGRET: A Reputation Model for Gregarious Societies," in *Proceedings of the 4th Workshop on Deception, Fraud and Trust in Agent Societies*, 2001, pp. 61–69.

[19] NICE, "http://www.cs.umd.edu/projects/nice/ (last accessed: Jan 09, 2012)."

[20] S. Banerjee, B. Bhattacharjee, and C. Kommareddy, "Scalable application layer multicast," *SIGCOMM Comput. Commun. Rev.*, vol. 32, no. 4, pp. 205–217, 2002.

[21] A. Crainiceanu, P. Linga, J. Gehrke, and J. Shanmugasundaram, "Querying peer-to-peer networks using P-Trees," in *Proceedings of the 7th WebDB*, 2004, pp. 25–30.

[22] Gnutella, "http://www.gnutella.com/ (last accessed: Jan 09, 2012)."

[23] Napster, "http://www.napster.com/ (last accessed: Jan 09, 2012)."

[24] BitTorrent, "http://www.bittorrent.com/ (last accessed: Jan 09, 2012)."

[25] Blue Frog, "http://sourceforge.net/projects/bluefrog/ (last accessed: April 12, 2006)."