Detecting Malicious Nodes in Peer-to-Peer Streaming by Peer-Based Monitoring

XING JIN
Oracle USA, Inc.
and
S.-H. GARY CHAN
The Hong Kong University of Science and Technology

Current peer-to-peer (P2P) streaming systems often assume that nodes cooperate to upload and download data. However, in the open environment of the Internet, this is not necessarily true and there exist malicious nodes in the system. In this article, we study malicious actions of nodes that can be detected through peer-based monitoring. We require each node to monitor the data received and to periodically send monitoring messages about its neighbors to some trustworthy nodes. To efficiently store and search messages among multiple trustworthy nodes, we organize trustworthy nodes into a threaded binary tree. Trustworthy nodes also dynamically redistribute monitoring messages among themselves to achieve load balancing. Our simulation results show that this scheme can efficiently detect malicious nodes with high accuracy, and that the dynamic redistribution method can achieve good load balancing among trustworthy nodes.

Categories and Subject Descriptors: C.2.0 [**Computer-Communication Networks**]: General—*Data communications*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Retrieval models; search process*

General Terms: Performance, Design

Additional Key Words and Phrases: Malicious nodes, peer monitoring, peer-to-peer streaming

ACM Reference Format:

Jin, X. and Chan, S.-H. G. 2010. Detecting malicious nodes in peer-to-peer streaming by peer-based monitoring. ACM Trans. Multimedia Comput. Commun. Appl. 6, 2, Article 9 (March 2010), 18 pages.
DOI = 10.1145/1671962.1671965 http://doi.acm.org/10.1145/1671962.1671965

1. INTRODUCTION

With the popularity of broadband Internet access, there has been increasing interest in media streaming. Recently, peer-to-peer (P2P) streaming has emerged to overcome limitations in traditional server-based streaming. In P2P streaming, cooperative peers self-organize themselves into overlay networks

This work was supported in part by the General Research Fund from the Research Grant Council of the Hong Kong Special Administrative Region, China (611107), the Cisco University Research Program Fund, a corporate-advised fund of Silicon Valley Community Foundation (SVCF08/09.EG01) and the Hong Kong Innovation Technology Fund (ITS/013/08).
Authors' addresses: X. Jin, Oracle USA, Inc., 400 Oracle Parkway, Redwood Shores, CA 94065; email: jin@oracle.com; S.-H. G. Chan, Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong; email: gchan@cse.ust.hk.

DOI 10.1145/1671962.1671965 http://doi.acm.org/10.1145/1671962.1671965

via unicast tunnels. They cache and relay data for each other, thereby eliminating the need for powerful servers from the system. Current P2P streaming systems have been shown to be able to support up to thousands of peers with acceptable quality of service [Zhang et al. 2005; Hei et al. 2007; Tang et al. 2007].

Many P2P streaming systems focus on improving streaming quality and assume that all nodes in the system cooperate as desired. However, this may not be true in the open environment of the Internet. Some nodes in the system may be selfish and unwilling to upload data to others. Some may have abnormal actions such as frequent rebooting, which adversely affect their neighbors. More terribly, some nodes may cheat their neighbors, launch attacks, or distribute viruses to disrupt the service. Following the notations in Jin et al. [2006a], we call this uncooperative, abnormal, or attacking behavior *malicious actions* and the corresponding nodes *malicious nodes*.

In this article, we study how to detect malicious nodes in a P2P streaming system. We focus on malicious nodes that degrade the streaming quality of their neighbors. If each node is monitored by its neighbors and the monitoring messages are collected to compute node reputation, nodes with malicious actions can be detected. We explore two important issues in this design.

—*How do we compute node reputation based on monitoring messages?* A node may lie about the performance of its neighbors in monitoring messages. Such a node can manipulate node evaluation as it desires, for example, increasing the reputation of malicious peers or decreasing the reputation of good peers. These lying nodes may not be easily detected, especially when they cache and relay streaming data as normal peers and are trusted by others.

To detect lying in monitoring messages, we employ a traditional approach for monitoring *suspicious messages* [Mekouar et al. 2006]. Each node is assigned a credit value based on the number of suspicious messages it generates, which indicates to what extend a later monitoring message from the node can be trusted. We then integrate node credit into reputation computing to address the node lying problem.

As each node periodically generates monitoring messages, the monitoring overhead is considerably high. We propose using linear prediction (LP) sampling [Hernandez et al. 2001] to adaptively adjust the frequency of generating monitoring messages. That is, if the neighbor of a peer is always well-behaved, the peer can reduce the frequency of generating monitoring messages about the neighbor. In this way, the monitoring overhead for well-behaved nodes can be reduced.

—*Where should the monitoring messages be stored and the reputation values be computed?* The storage of monitoring messages and the retrieval of reputation values are not trivial. If messages are stored at some normal nodes, these nodes may modify or forge data in messages. Similarly, if a reputation value is computed by a normal node, we are not aware whether the value has been manipulated or not. A straightforward solution is to use a dedicated server to collect monitoring messages and to compute node reputation. However, given thousands of nodes in the system (which is often the case in current P2P streaming systems), a single server may be easily overloaded and form a single point of failure.

In our approach, we use a monitoring overlay formed by a set of trustworthy nodes to manage monitoring messages. We assume that these trustworthy nodes have been certificated and will not modify or forge monitoring messages or reputation values. For example, they could be a set of predeployed proxies. Monitoring messages are stored at these trustworthy nodes and reputation is also computed by these trustworthy nodes. In the following, we call a trustworthy node in the monitoring overlay a *monitoring node*, and a normal node in the streaming overlay a *streaming node*. We study how to organize monitoring nodes to ensure that an update or query to node reputation can be quickly accomplished. We also study how to allocate monitoring loads on monitoring nodes to achieve load balancing among them.

We have conducted simulations to evaluate our scheme. The results show that it can detect malicious nodes with low false positive and false negative rates. And LP sampling can significantly reduce monitoring overhead, especially when the portion of malicious nodes is small. Furthermore, the loads on monitoring nodes can be efficiently balanced by our dynamic redistribution method.

In the following, we assume that most streaming nodes in the system are well-behaved, and that node behavior is consistent for a considerably long time. The rest of the article is organized as follows. In Section 2 we discuss related work. In Section 3 we describe reputation computing based on monitoring messages. In Section 4 we discuss the structure of monitoring nodes and the load redistribution mechanism. In Section 5 we present illustrative simulation results. Finally, we conclude in Section 6.

2. RELATED WORK

In this section, we briefly review previous work on P2P streaming and P2P reputation.

2.1 Peer-to-Peer Streaming

Multimedia applications (e.g., on-demand video streaming and IPTV) often involve media streaming from a source to a large population of users. A traditional delivery technique for these applications is IP multicast. In IP multicast, routers are interior nodes in a multicast tree and are responsible for forwarding data down the tree, while all end hosts are leaves in the multicast tree [Deering 1988]. An advantage of IP multicast is that a single channel can be shared by infinitely many users. It hence makes efficient use of network bandwidth. However, IP multicast requires multicast-capable routers, which are not widely available in the current Internet infrastructure. Furthermore, IP multicast lacks large-scale commercial management functions such as multicast address allocation and reliable transmission. Therefore, although IP multicast has been proposed for over ten years, it has not yet been widely deployed. Another delivery technique is the client/server transmission mode, which allocates network resources for each specific client request. However, due to heavy load and limited network bandwidth at the server side, this approach does not scale well.

Recently, P2P streaming has been proposed and developed to overcome limitations in IP multicast and server-based streaming. In P2P streaming, cooperative peers self-organize into an overlay network via unicast tunnels. They cache and relay data for each other, thereby eliminating the need for powerful servers from the system. Clearly, P2P streaming uses only unicast and does not need multicast-capable routers.

Currently, there are two types of overlays for P2P streaming: tree structure and gossip mesh. The first builds one or multiple overlay trees to distribute data among peers. Examples include application-layer multicast protocols (e.g., Narada [Chu et al. 2002] and NICE [Banerjee et al. 2002]) and some P2P video-on-demand systems (e.g., P2Cast [Guo et al. 2003] and P2VoD [Do et al. 2004]). The second builds a mesh among peers using gossip algorithms, with peers exchanging data with their neighbors in the mesh [Zhang et al. 2005; Tang et al. 2007]. In spite of their high control overhead, mesh overlays are resilient in the face of changing network and group dynamics. On the other hand, trees introduce lower end-to-end delay and are easier to maintain. But in tree-based approaches, failure or packet loss at a peer may affect all its descendants. A tree is hence not resilient in the face of peer and network errors.

Our scheme in this article does not impose any restrictions on streaming overlays, and can be applied to any streaming overlays.

2.2 Peer-to-Peer Reputation

Due to their self-organizing nature, most P2P networks lack strict control for joining nodes. Malicious nodes can easily join and reside in P2P networks. To detect these malicious nodes and reduce their

impacts on the system, a reputation system for nodes is often used. In this section, we briefly review reputation computing and storage issues in P2P networks.

2.2.1 *Reputation Computing.* Many reputation systems compute reputation as in a social network. All feedbacks available in the network are aggregated for reputation computing. This can be classified into two categories: *separated reputation model* and *correlated reputation model*. In a separated reputation model, only the direct transaction partners (e.g., resource providers/ downloaders or streaming neighbors) of a peer can express their opinion on the reputation of the peer [Cornelli et al. 2002; Damiani et al. 2002; Dragovic et al. 2003; Singh and Liu 2003; Jun et al. 2005; Mekouar et al. 2006; Jin et al. 2006a]. Our reputation model in this article also belongs to this category.

A practical example of a separated reputation model is the eBay reputation system (although eBay is not a P2P network).[1] After each transaction at eBay, the buyer and the seller rate each other with positive, negative, or neutral feedback. The reputation is calculated at a central server by assigning 1 point for each positive feedback, 0 point for each neutral feedback and -1 point for each negative feedback. The reputation of a participant is computed as the sum of its points over a certain period. Considering that peers may not be truthful in their feedback, Mekouar et al. [2006] propose monitoring suspicious feedback. The more suspicious feedback a peer generates, the smaller its weight in computing reputation. We adopt a similar approach in this article. Xiong and Liu [2004] develop a general reputation model, which considers, for example, feedback from other peers, credibility factor for the feedback sources, and transaction context factor for discriminating the importance of transactions. Almost all the separated reputation models can be expressed by this generalized model.

In a correlated reputation model, the reputation of a peer is computed based on the opinions of its direct transaction partners as well as some third-party peers [Kamvar et al. 2003; Sherwood et al. 2006]. In this model, a peer A, who wishes to know the reputation of another peer B, can ask some peers (e.g., its neighbors) to provide their opinion of B (although some of the peers may not have conducted any transactions with B). A then combines the opinions from the peers to calculate B's reputation. This model is more like our real social networks, where third-party peers other than transaction partners can express their opinions on a peer. But it costs more to collect and aggregate third-party opinions.

2.2.2 *Reputation Storage and Retrieval.* A basic principle in reputation storage is that the reputation of a peer cannot be locally stored at the peer. Because this will not afford protection against dishonest peers. A dishonest peer may misreport its reputation value in order to gain rewards or avoid punishment.

The simplest solution is to use a powerful server to keep the reputations of all peers. For example, eBay uses a central server to collect and keep all users' reputations. Feedback from users is sent to and stored at the server. A query of a user's reputation is also sent to and answered by the server. Similar approaches have been used in Dragovic et al. [2003]; Jun et al. [2005]; and Jin et al. [2006a]. This approach is easy to implement and deploy. Centralization makes reputation management independent of peers joining and leaving, which greatly simplifies reputation retrieval. However, a centralized approach is not scalable to large P2P networks. In addition, the server forms a single point of failure, making the system vulnerable.

To address the limitations of the centralized approach, a partially centralized approach that uses a set of servers instead of a single server has been proposed. Mekouar et al. [2006] propose a malicious detector algorithm (MDA) to detect malicious peers in KaZaa-like systems. KaZaa[2] is a partially centralized P2P search system with a set of supernodes. MDA assumes that supernodes are all trustworthy and maintain reputation information for ordinary peers. Each peer is attached to a unique supernode.

[1]http://www.ebay.com
[2]http://www.kazaa.com

All evaluation results about a peer are maintained at its attached supernode. Supernodes can then enforce differentiated services to peers according to their reputations.

Our scheme also adopts the partially centralized structure. We use a set of trustworthy nodes (e.g., proxies). Each streaming node is attached to a unique trustworthy node. A streaming node periodically sends monitoring reports about its streaming neighbors to trustworthy nodes. A query about a streaming node's reputation is forwarded to and answered by the node's attached trustworthy node.

Two important issues in the partially centralized approaches are efficient search and load balancing among multiple supernodes/proxies. First, we need to attach each peer to a unique supernode (or proxy). In MDA, this is done by a KaZaa built-in mechanism. In a more general P2P network without supernodes, this is not easy. Suppose each proxy is responsible for a certain range of peers. Given any peer in the system, we need to quickly identify the proxy responsible for it (e.g., for reputation update or query). If the number of proxies is small, simple flooding can be used for search. Otherwise, a more complicated overlay structure should be built among proxies. In our scheme, proxies maintain disjoint IP ranges and organize themselves into a binary search tree according to their IP ranges. Based on the tree, our approach can achieve high search efficiency. Second, the loads for reputation management should be evenly distributed among supernodes/proxies. Otherwise, some supernodes may be quickly overloaded. MDA does not consider the load balancing issue as it uses the KaZaa built-in mechanism to attach peers to supernodes. On the contrary, our scheme employs a dynamic load redistribution method to balance the loads among proxies. Our simulation results have shown that this method can effectively balance the proxy loads.

There are also some distributed approaches for reputation storage in P2P systems. For example, some use distributed hash table (DHT) systems to store and search node reputation [Aberer and Despotovic 2001; Kamvar et al. 2003; Xiong and Liu 2004]. Others use unstructured overlays to track and share information about the reputation of nodes and resources [Cornelli et al. 2002; Damiani et al. 2002]. Although these approaches are fully distributed and scalable, they have some security limitations. First, during the search or delivery of peer reputation, an uncooperative or malicious peer in the delivery path may modify, intercept, or discard the message. Second, a feedback provider may misbehave by providing false data or random data when responding to a query. As a comparison, our scheme uses trustworthy nodes to manage reputation. We organize trustworthy nodes into a binary search tree so that the search and storage operations can be quickly accomplished.

The preliminary version of this work has appeared in Jin et al. [2006b], where we study how to compute peer reputation and how to manage reputation for efficient storage and search. In this article, we extend it by adding LP sampling to reduce monitoring overhead and using an AVL-tree structure [Knuth 1998] to improve search efficiency.

3. DETECTING MALICIOUS ACTIONS

3.1 Malicious Actions in P2P Streaming

We are interested in malicious actions that can be detected through the monitoring of nodes' past performance. The following are a few examples of such actions.

—*Attacking actions*. Some nodes intend to degrade system performance and disrupt service. Their attacking actions include:

(1) *Eclipse attack*. In an overlay network, if an attacker controls a fraction of the neighbors of a legitimate node, it can eclipse the legitimate node by dropping or rerouting messages, thus adversely affecting proper overlay operations [Singh et al. 2004]. As indicated in Singh et al. [2004], such an attacker usually has much larger indegree and outdegree than normal peers. We can hence require a node to report its connections with its parents. For a certain node, if all its children

report a connection with it, we can accordingly compute the outdegree of the node. Similarly, we can compute node indegree.

(2) *Denial of service (DoS)*. DoS attacks involve the adversary bringing to bear large amounts of resources in order to completely disrupt service usage [Marti and Garcia-Molina 2006]. A node may unfairly increase its access to shared resources, for example, requesting as much data as possible from as many peers as possible [Nielson et al. 2005]. To detect this, a node may report the amount of resources it provides to each of its neighbors, for example, the amount of downloading data, the duration or bandwidth of the downloading connection. The total amount of resources that a node is consuming can then be analyzed given all its neighbors' reports.

(3) *Distributing corrupt data*. A node may distribute corrupt data that do not conform to the stream format [Jun et al. 2005]. To detect it, a node may report the amount of corrupt data its parent sends with respect to the total amount of data the parent sends.

—*Abnormal behavior*. Some nodes do not intend to attack the system, but their actions adversely affect other peers and degrade the service. These abnormal actions include:

(1) *Frequent joining/leaving*. P2P systems are highly dynamic due to their self-organizing nature. It has been observed that most users in a P2P file sharing system spend only minutes per day browsing the network while a handful of other peers exhibit server-like behavior and keep their computers logged in for weeks at a time [Lai et al. 2003]. In a streaming system, if a node unexpectedly leaves the system, its children have to search for new parents. These nodes will experience service outage before receiving data from new parents. To detect abnormal joining or leaving of a node, we require the neighbors of the node to report the joining and leaving times of the node. The average online time of the node can then be computed.

(2) *Free riding*. A node may only download data but not share them with others [Adar and Huberman 2000]. These nodes are usually called selfish nodes. They use system services while contributing minimal or no resources themselves. This can be detected by a node reporting the amounts of download and upload data of a neighbor.

Note that a reputation system has limitations. For example, peers may purposefully leave and rejoin the system with a new identity in an attempt to shed any bad reputation they have accumulated under their previous identity. These peers are called whitewashers [Lai et al. 2003]. Multiple malicious peers may also collude to increase their own reputation [Marti and Garcia-Molina 2006]. Detecting and defending against these specific attacks to reputation systems are independent research topics. Interested readers can refer to Marti and Garcia-Molina [2006] for an overview.

3.2 Reputation Computing

We define a *submission period* as a certain time duration. Each node needs to generate monitoring messages to report the performance of its neighbors in each submission period. We now study how to compute the reputation of a streaming node A, $REP(A)$, from monitoring messages. For ease of illustration, we take one of the malicious actions, distributing corrupt data, as an example. That is, a node may send corrupt data that do not conform to the stream format.

Note that a dishonest streaming node may submit forged monitoring messages to affect reputation computing. To address this problem, we monitor *suspicious messages* as in Mekouar et al. [2006]. When a data receiver submits a monitoring message, the corresponding data sender also submits its own version of the message to describe the data delivery. If there is an obvious gap between these two messages, both messages are regarded as suspicious. Note that the gap depends on the reputation model. For example, if we are detecting the distributing of corrupt data, we can compare the amounts of corrupt data reported in the two messages. If the two reported amounts differ by a certain threshold

(which may be set according to real measurement data), we regard the messages as suspicious. If we are detecting an eclipse attack, we need nodes to report their connections with parents. If one message reports a connection between the nodes while the other one claims no connections between them, the two messages are both suspicious.

Define $N_s(A)$ and $N_n(A)$ as the number of suspicious messages and non-suspicious messages generated by A. As in Mekouar et al. [2006], the credibility level of A, $Credit(A)$, is given by $N_n(A)/(N_s(A) + N_n(A))$. It indicates to what extend we should believe the messages generated by A. Since dishonest nodes often have low credit values, their opinion on the evaluation of others should be accordingly reduced. We hence compute $REP(A)$ as:

$$REP(A) = Average\left\{Credit(X) \times \left(1 - \frac{Corrupt(A, X, t)}{Total(A, X, t)}\right), \forall\, t \in valid\ periods;\right\},$$

where $Corrupt(A, X, t)$ and $Total(A, X, t)$ are the amounts of corrupt data and total data received by X from A in period t as reported by X, respectively. The valid periods exclude outdated messages. The *Average* function computes the average value over all available X.

For each of the malicious actions in Section 3.1, we can independently define node reputation. We show another example of how to detect an eclipse attack. As discussed, the indegree of an attacker is always higher than the average node indegree. Therefore, legitimate nodes can choose their neighbors from the overlay nodes whose indegree is lower than a certain threshold. In addition, attacking nodes may consume the indegree of legitimate nodes and prevent other legitimate nodes from pointing to them. Thus, it is also necessary to bound node outdegree. In summary, the indegree and outdegree of nodes should be smaller than certain thresholds. Nodes violating this constraint will be regarded as attacking nodes. In order to detect an eclipse attack, we require each node to periodically report its connection to the parent. In other words, we can compute node reputation as follows:

$$\begin{aligned}
REP_{indegree}(A, t) &= Sum\{Credit(X) \times F(A, X, t)\},\\
\text{where } F(A, X, t) &= \begin{cases} 1 & \text{if } X \text{ reports } A \text{ as the child during period } t;\\ 0 & \text{Otherwise.}\end{cases}\\
REP_{outdegree}(A, t) &= Sum\{Credit(X) \times F'(A, X, t)\},\\
\text{where } F'(A, X, t) &= \begin{cases} 1 & \text{if } X \text{ reports } A \text{ as the parent during period } t;\\ 0 & \text{Otherwise.}\end{cases}
\end{aligned}$$

If a node's $REP_{indegree}$ is higher than a certain threshold, or its $REP_{outdegree}$ is higher than another threshold, we classify the node as an attacking node.

3.3 Reputation-Based Peer Selection

Reputation values are used to guide peer selection during streaming. A higher reputation value indicates that the node is more trustworthy in terms of the collective evaluation by peers that have had data exchange with it. There are various ways to use reputation values. For example, a peer that issues downloading requests may receive several responses. It can compare the reputation of the responding peers and choose the one with the highest reputation to download data. This reduces the risk of receiving abnormal service from malicious nodes. XREP has adopted this approach [Damiani et al. 2002; Cornelli et al. 2002]. In another example, if node A wants to set up a connection with node B, A first queries B's reputation. If B has low reputation (i.e., a potential malicious node), A blacklists B and does not connect to it for a while. This is similar to the eBay reputation system. In another case, if some node is considered as malicious, say because its reputation is lower than a certain threshold,

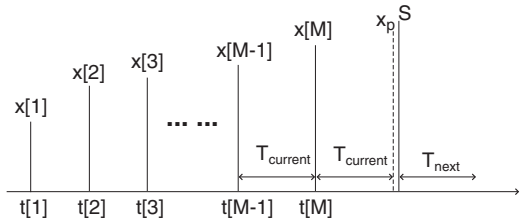


Fig. 1. Linear prediction sampling.

its attached monitoring node can broadcast this information to other peers so that they may block it.

We can also develop incentive mechanisms based on node reputation. Many P2P streaming systems have employed differentiated service to motivate node sharing. For example, in Habib and Chuang [2006], peers are evaluated based on their contribution to others. A peer with high contribution will obtain a relatively high rank, and each peer is allowed to select peers with equal or lower rank as streaming parents. A peer with a higher rank hence has more parent candidates and is likely to achieve better streaming quality. In another example, peers earn points by forwarding data to others, and bid for good parents using their points [Tan and Jarvis 2006].

3.4 Reducing Monitoring Overhead

In this design, a node needs to periodically generate and send monitoring messages about its neighbors. Since every node is monitored, the total monitoring overhead is high, especially in a large-scale system. To reduce it, we use linear prediction (LP) sampling [Hernandez et al. 2001] to adaptively adjust the frequency of generating monitoring messages.

We illustrate the idea of LP sampling in Figure 1. Vector x holds the values of the previous M samples, where $x[M]$ is the latest sample and $x[1]$ is the oldest. In our case, a sample is the ratio of the amount of corrupt data to the amount of total data in a certain submission period. Vector t records the time that each sample in x is taken (in the unit of submission period in our example). $\Delta T_{current}$ is the time gap between $t[M-1]$ and $t[M]$. Variables x_p and S represent the predicted and actual values of the sample taken at time $t[M] + \Delta T_{current}$, respectively, where x_p is computed as follows:

$$x_p = x[M] + \Delta T_{current} \frac{1}{M-1} \sum_{i=1}^{M-1} \left(\frac{x[i+1] - x[i]}{t[i+1] - t[i]} \right).$$

A set of rules is applied to adjust the current sampling interval, $\Delta T_{current}$, to a new value, ΔT_{next}, which is used to schedule the next sampling. Define $m = (x_p - x[M])/(S - x[M])$. The new sampling interval is computed as:

$$\Delta T_{next} = \begin{cases} \lfloor m \times \Delta T_{current} \rfloor & if\ m < m_{min}; \\ \Delta T_{current} & if\ m_{min} \le m \le m_{max}; \\ \Delta T_{current} + 1 & if\ m > m_{max}; \\ 2 \times \Delta T_{current} & if\ S = x[M]. \end{cases}$$

Based on the results of the simulations described later, it is good to set m_{min} and m_{max} to 0.7 and 1.2, respectively. An additional bound of [1, 10] is used to limit the range of ΔT_{next}. The lower bound of 1 indicates the minimum time interval to schedule the next sampling (one submission period), while the upper bound ensures that there is a certain minimum set of samples on which to base future predictions. Note that given a certain i, there are no samples in the periods between $t[i]$ and $t[i+1]$

(suppose $t[i+1] > t[i]+1$). It is inherently assumed that the values in these periods are equal to $x[i]$.

In our scheme, LP sampling works as follows. Suppose that node A is monitoring node B. A determines the frequency of generating monitoring messages according to B's performance. Each time it sends a monitoring message, it notifies B to accordingly generate a monitoring message. These messages are then compared at monitoring nodes to detect lying.

4. REPUTATION STORAGE AND RETRIEVAL

In this section, we discuss reputation storage and retrieval at monitoring nodes.

4.1 Design Overview

We consider a set of trustworthy nodes (called monitoring nodes) in the system. A trustworthy node always behaves as desired never modifies or forges data. Trustworthy nodes form a monitoring overlay to store and maintain monitoring messages from streaming nodes. Each trustworthy node holds a certificate issued by a trusted certification authority. With the certificates, two trustworthy nodes can authenticate each other and set up a secure connection as in secure socket layer (SSL) [SSL]. A trustworthy node can be a predeployed proxy or an authenticated end host. Selection of trustworthy nodes is beyond the scope of this article. Interested readers can refer to Chen and Yeager [2001].

In our scheme, each streaming node, P, is attached to a monitoring node, M_P. All messages about the performance of P are forwarded to M_P. The reputation and credit of P are then computed and maintained by M_P. A streaming node can send two types of messages to the monitoring overlay

—*UPDATE message*. A streaming node keeps checking the data it receives and sends an *UPDATE* message to report the performance of its neighbor when necessary. An *UPDATE* message contains the data sender's IP address, the data receiver's IP address, information about the data sender's performance (e.g., the amount of corrupt data detected) and a timestamp. Here we assume that a streaming node can be uniquely represented by its IP address.

  As mentioned in Section 3.2, both a data sender and a data receiver need to send monitoring messages in order to detect suspicious messages. Consider a sender S and a receiver R. Suppose that they are attached to monitoring nodes M_S and M_R, respectively. Both S and R will keep sending *UPDATE* messages (which describe the performance of S) to the monitoring node M_S. After M_S analyzes all the monitoring messages about S, M_S can accordingly compute the reputation and credit of S. Note that since R has sent monitoring messages about S, M_S also needs R's credit value when evaluating S, which is maintained by M_R. Therefore, M_S needs to search for M_R in the monitoring overlay (using the search method introduced in Section 4.3) before evaluating S.

—*QUERY message.* A streaming node can send a *QUERY* message to query the reputation of any other streaming node.

On the other hand, a monitoring node, A, is responsible for a certain IP range $[A_l, A_r)$. A streaming node with IP address within this range will be attached to A. Clearly, IP ranges maintained by monitoring nodes should not overlap. We further define $L(A)$ as the load on a monitoring node A: the number of IP addresses that have been attached to A.

4.2 Construction of the Monitoring Overlay

To efficiently process messages from streaming nodes, we organize monitoring nodes into a *threaded binary tree*. A threaded binary tree is a binary search tree in which each node maintains a *Pred* link pointing to the node's in-order predecessor and a *Succ* link pointing to its in-order successor [Cormen et al. 2001]. Each node should also maintain two links pointing to its left child and right child. Figure 2

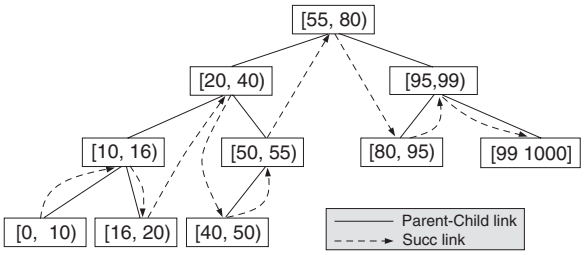


Fig. 2. An example of a threaded binary search tree.

Table I. Fields at Each Node in the AVL Tree

Field	Meaning
Parent	Parent in the tree
Grandparent	Parent's parent in the tree
LeftChild	Left child in the tree
RightChild	Right child in the tree
LeftHeight	Height of the left subtree
RightHeight	Height of the right subtree
Pred	In-order predecessor
Succ	In-order successor

shows an example of a threaded binary tree. Each quadrangle in the figure is a monitoring node, and the numbers in a quadrangle indicate the IP range maintained by the node. Here we use numerical values to represent IP addresses. Suppose that a node K is responsible for the IP addresses in the half-open interval $[K_l, K_r)$. The ranges maintained by nodes in the left subtree of K are all smaller than K_l, and the ranges maintained by nodes in the right subtree of K are larger than or equal to K_r. The directed dash lines in the figure indicate the in-order successors of nodes. Starting from the leftmost leaf node of the tree (the node responsible for [0, 10) in the figure) and following the dashed lines, all the nodes can be traversed in an ascending order according to their ranges. The *Pred* links are not shown in the figure, since they can be tracked through the reverse of the *Succ* links. That is, if node A is node B's successor, B is A's predecessor.

The height of a binary tree can be as large as the number of nodes in the tree. To improve search efficiency, we maintain an AVL search tree. An AVL tree is a binary search tree with the following property: for any node in the tree, the heights of its left and right subtrees can differ by at most 1 [Knuth 1998]. It is known that, for an AVL tree with N nodes, its height H satisfies $H < 1.44 \log(N+2) - 0.328$. Hence, the cost of locating a certain monitoring node is $O(\log N)$ in a tree of N monitoring nodes. We list in Table I, the information kept at each monitoring node in the AVL tree.

4.2.1 *Node Joining.* A new monitoring node maintains an empty IP range. It is inserted into the tree as follows. The new node first contacts the tree root. If the *LeftHeight* of the root is larger than the *RightHeight* of the root, the root redirects the new node to its right child in the tree. Otherwise, the root redirects the new node to its left child. This process is repeated until the corresponding child is empty. The new node is then inserted to this position as a leaf node.

If the new node is inserted as the left child of its parent, it will become the predecessor of its parent. Meanwhile, the old predecessor of the parent becomes the predecessor of the new node. Similarly, if the new node is inserted as the right child of its parent, it becomes the successor of its parent, and its own successor is the old successor of the parent.

The new node then sets its height to 0, and sends a *HeightReport* message to its parent in the tree. Upon receiving the message, the parent updates its *LeftHeight* or *RightHeight*, depending on which branch the report comes from, and then calculates its own height as $max(LeftHeight, RightHeight) + 1$. If the height value changes, the node continues reporting its height to its own parent until the tree root is reached.

Following this joining mechanism, tree balance will not be violated, and no subtree rotation is needed. Since the height of an AVL tree is $O(\log N)$, the cost of a joining operation is bounded by $O(\log N)$.

Note that all node joining starts from the tree root, which is similar to many existing overlay tree protocols (e.g., Overcast [Jannotti et al. 2000], NICE [Banerjee et al. 2002], HMTP [Zhang et al. 2002] and Zigzag [Tran et al. 2004]). The root should then have enough computational and communication resources. To alleviate root load and prevent possible root failure, we can use backup roots as in Overcast.

4.2.2 *Node Leaving.* Nodes in the tree periodically exchange *KeepAlive* messages with their parents and children. Failure or leaving of a node can be detected because its *KeepAlive* messages will be missing. The recovery operations after failure or leaving of a node are much more complicated than node joining, mainly due to subtree rotation for tree balancing. In Liu and Zhou [2006], a distributed approach has been proposed to recover node failure in an AVL tree. This approach can be directly applied to our scheme. We hence omit the details of node leaving here. Interested readers can refer to Liu and Zhou [2006].

4.3 Access and Maintenance of Reputation

A message is processed in the monitoring overlay, as Figure 3 shows. Suppose that streaming node B is streaming node A's child in the streaming overlay, and B prepares to submit an *UPDATE* message to report A's performance. If B has never sent an *UPDATE* message about A before, B first sends A's IP address to a random monitoring node, which is R in the example. R then searches in the threaded tree to identify the monitoring node whose range covers A's IP address (node T in this case). T then sends a response message to B along with its certificate of trustworthiness. After B verifies the trustworthiness of T, it sends its *UPDATE* message about A to T. In the following periods, B will directly send the *UPDATE* messages about A to T.

The IP ranges maintained by monitoring nodes should be carefully allocated so that their loads are balanced. Initially, the root of the tree maintains the whole IP range while all the others maintain an empty range. A monitoring node, A, periodically compares its own load with the loads of its predecessor A_- and successor A_+ (if any). If the gap between $L(A)$ and $L(A_-)$ or the gap between $L(A)$ and $L(A_+)$ is larger than a certain threshold δ, the three nodes redistribute their loads and IP ranges to equally share the loads. If a monitoring node leaves the tree, its load is distributed to its predecessor and successor before it leaves. Note that with unexpected node leaving or failure, the data maintained at the node are no longer available. This is a common issue in distributed storage systems [Rowstron and Druschel 2001; Stoica et al. 2001]. Traditional solutions include erasure coding and replication [Rodrigues and Liskov 2005]. For example, in a typical peer-to-peer storage system, PAST, each data file has five replicas in the system [Rowstron and Druschel 2001]. In our system, we may similarly require each node to send a copy of its data to its predecessor and successor and periodically update them.

Figure 4 shows an example of load redistribution among nodes. Initially, all the IP addresses are maintained by the root R, and all other nodes have empty ranges. With the insertion of new IPs, when R finds that its load $L(R)$ is larger than its successor R_+'s load, $L(R_+)$ (or its predecessor R_-'s load $L(R_-)$), by δ, R moves one-third of its load to R_- and another one third to R_+. R further redistributes the ranges among the three nodes as Figure 4(b) shows.

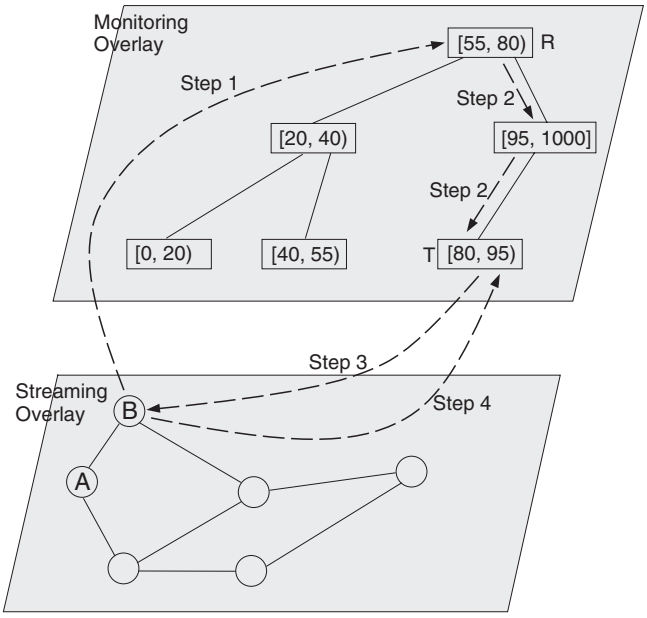


Fig. 3. The process of submitting an *UPDATE* message about a streaming node A by its child B for the first time. Step (1) B sends A's IP address to R. Suppose that A's IP address is represented by a numerical value 88. Step (2) R searches in the threaded binary tree to identify the monitoring node that manages 88 (node T in this case). Step (3) T responds to B with its certificate. Step (4) After verifying the trustworthiness of T, B sends its *UPDATE* message about A to T.

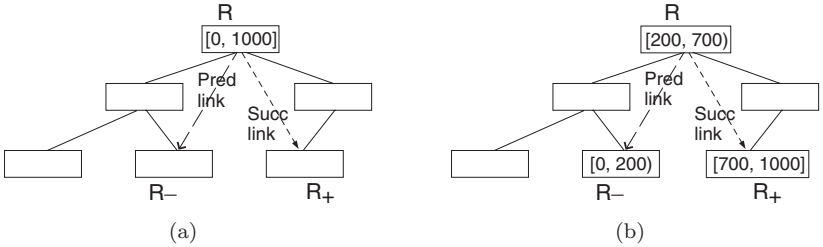


Fig. 4. Load and range redistribution among monitoring nodes. (a) Initially, root R maintains the whole IP range. (b) R redistributes its load and range to the predecessor R_- and the successor R_+.

With the redistribution of loads on monitoring nodes, streaming nodes should also be notified of the change in some way. Suppose a streaming node A is previously attached to a monitoring node M_{A1}, and now attached to M_{A2} after load redistribution. If a monitoring message about A from another streaming node, say B, has been generated, the message will be sent to M_{A1} as usual. Since M_{A1} knows that it is not responsible for A any more, it searches for the node responsible for A in the monitoring overlay, which should be M_{A2}. M_{A1} then forwards the monitoring message to M_{A2} and notifies B that M_{A2} is now responsible for A. Meanwhile, M_{A2} will contact B and set up connections with B as in Steps 3 and 4 in Figure 3.

5. ILLUSTRATIVE NUMERICAL RESULTS

In this section we present simulation results on Internet-like topologies.

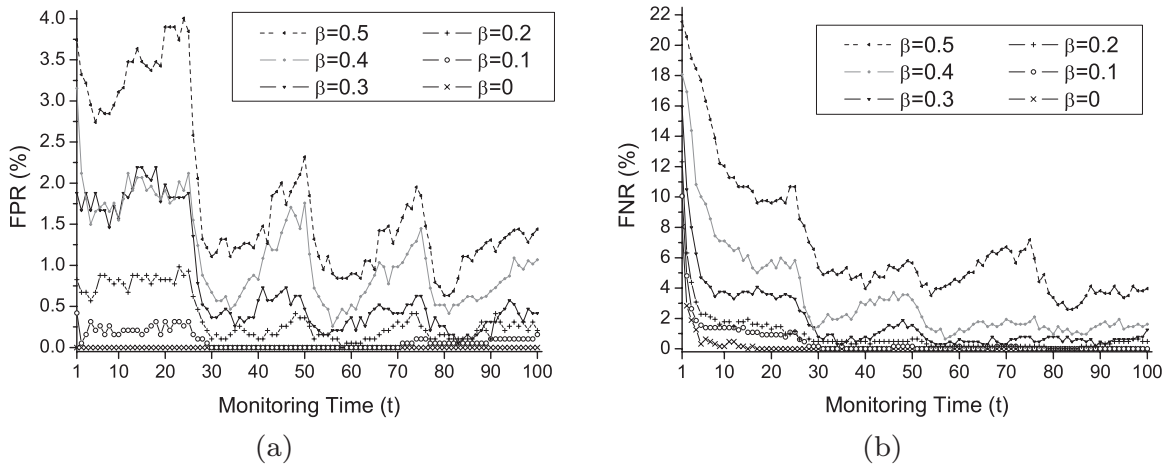


Fig. 5. Performance with different lie ratios (with group size 5000). (a) FPR. (b) FNR.

5.1 Simulation Setup

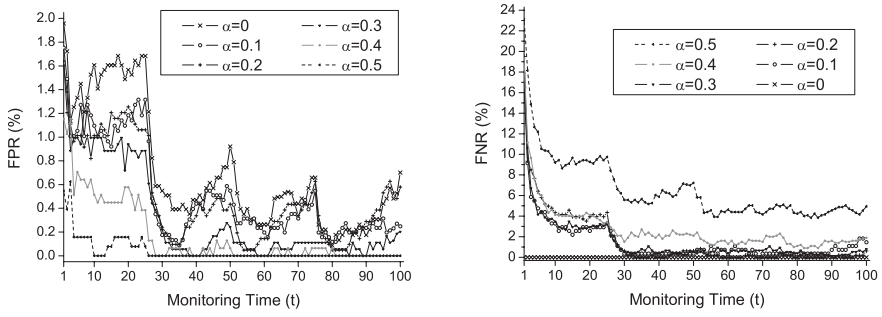
We randomly put a group of (1000–9000) streaming nodes into the network. Nodes form a streaming overlay as follows. Each node randomly selects multiple nodes as its parents, and a node can have at most 10 children. A node may also dynamically change some of its parents. The average duration of a streaming connection is 25 submission periods.

We simulate the example of distributing corrupt data. A non-malicious streaming node distributes a negligible amount of corrupt data, while a malicious streaming node distributes corrupt data with a probability uniformly distributed between [0.4, 1]. We define *malice ratio (α)* as the number of malicious streaming nodes divided by the total number of streaming nodes. Furthermore, a node may lie in its *UPDATE* messages. We evaluate three types of lies in our simulations. (I) The node monitored is not malicious, but its dishonest neighbor reports its action as malicious. (II) The node monitored is malicious, but its dishonest neighbor reports its action as nonmalicious (collusion). (III) A dishonest and malicious node reports its own action as nonmalicious. In the system, a streaming node is either honest or dishonest. An honest node never lies in its messages. For dishonest nodes, the type-I and type-III lies occur with a probability uniformly distributed in [0.4, 1], and the type-II lies occur with probability 0.05. We define *lie ratio (β)* as the number of dishonest streaming nodes divided by the total number of streaming nodes. Note that the behaviors of malice and lying are independent.

A node is evaluated as malicious if its reputation is smaller than the average reputation value of all the non-leaf nodes and a given threshold 0.85.

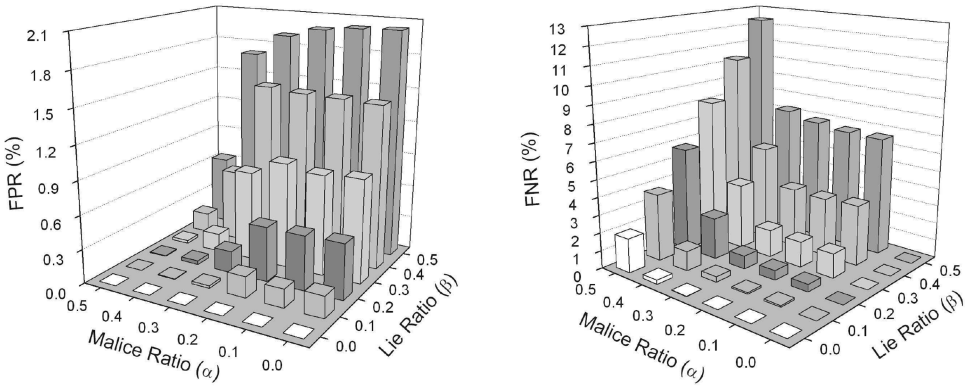
5.2 Results

We define *false positive rate (FPR)* as the number of non-malicious nodes evaluated as malicious divided by the total number of non-malicious nodes, and define *false negative rate (FNR)* as the number of malicious nodes evaluated as non-malicious divided by the total number of malicious nodes. Figure 5 shows *FPR* and *FNR* values versus monitoring time (in the unit of submission period) given different lie ratios, where the group size (the number of streaming nodes) is set to 5000. In the first several dozen periods, both *FPR* and *FNR* are large. They quickly decrease with the monitoring time. It shows that in the starting stage the credit and reputation values are skewed. The monitoring duration does not need to be long (about 30 periods) to achieve good enough accuracy. The larger lie ratio, the larger *FPR* and



(a) (b)

Fig. 6. Performance with different malice ratios (with group size 5000). (a) FPR. (b) FNR.



(a) (b)

Fig. 7. Performance with different malice ratios and lie ratios (after 30 submission periods, with group size 5000). (a) FPR. (b) FNR.

FNR. In Figure 5(a), *FPR* is kept below 2.5% in the steady stage, even with 50% dishonest nodes. In the best case of $\beta = 0$, *FPR* is always 0. On the other hand, we suffer much larger *FNR* values, as shown in Figure 5(b). This is partially due to the low evaluation threshold we set. If we increase the threshold, FPR will increase and FNR will decrease. As it is more important to protect nonmalicious nodes than to detect malicious ones, we set a low threshold to achieve low FPR, but relatively high FNR.

Figure 6 shows *FPR* and *FNR* values with different malice ratios. Similarly, they are large in the starting stage and become much smaller in the steady stage. We note in Figure 6(a) that the scheme achieves higher *FPR* when the malice ratio is low. When $\alpha = 0$, the *FPR* value is almost the largest. This is because our judgment is based on the average reputation value. With a small malice ratio, a large portion of nodes are non-malicious and have similar reputation values, which are also close to the average. In this case, a small perturbation to the evaluation may lead to an incorrect judgment. While in the case of large malice ratios, the gap between the average reputation and the reputation of a nonmalicious node is large, therefore nonmalicious nodes are unlikely to be evaluated as malicious.

Figure 7 shows the FPR and FNR values after 30 submission periods with different lie ratios and malice ratios. In general, when lie ratio β is large, *FPR* becomes large, especially when malice

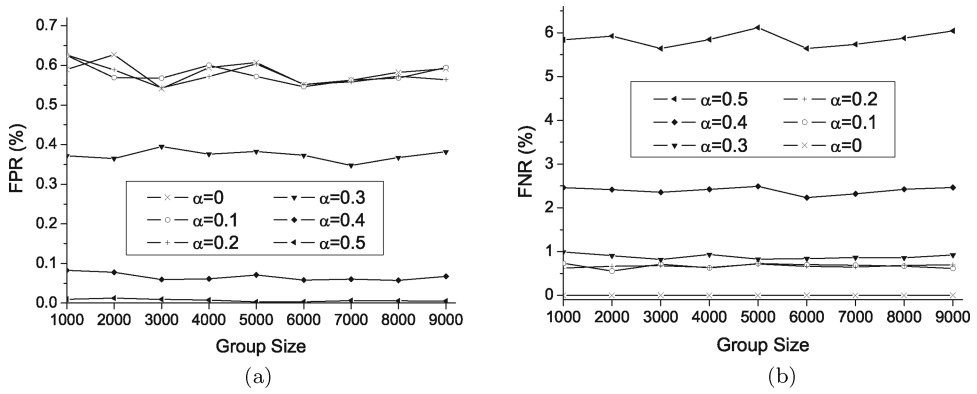


Fig. 8. Performance with different group sizes (after 30 submission periods, $\beta = 0.25$). (a) FPR. (b) FNR.

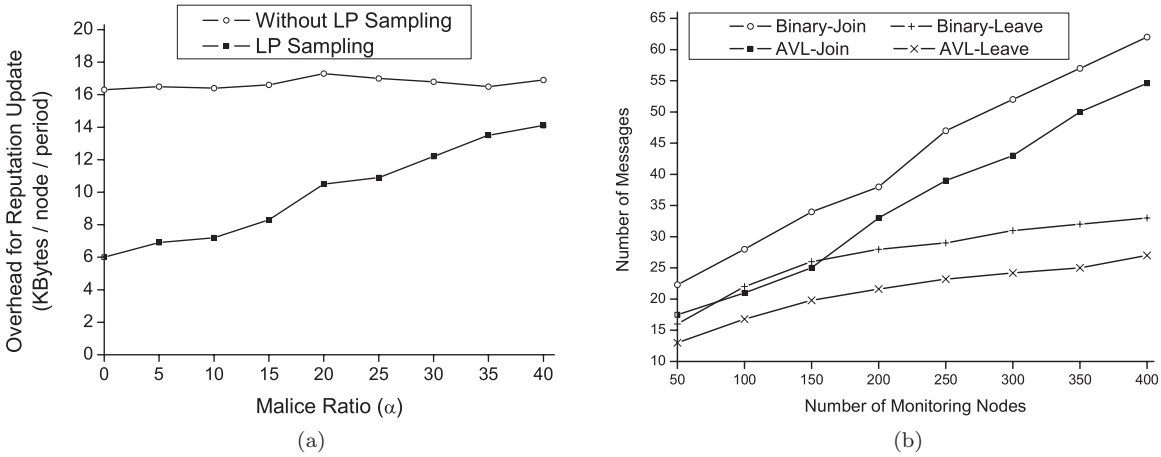


Fig. 9. Control overhead in the system. (a) Overhead for reputation update. (b) Overhead for maintaining the monitoring overlay.

ratio α is small. But for *FNR*, the worst performance is achieved when both α and β are the largest.

Figure 8 shows the FPR and FNR values with different group sizes. The FPR and FNR values are almost independent of the group size. This is because a node's reputation is determined by its neighbors, and the selection of neighbors in our simulations does not depend on group size. In Figure 8(a), the FPR value is large when the malice ratio is small. The reason has been explained before. Similar to Figure 7, the scheme reaches a much larger FNR than FPR (as shown in Figure 8(b)).

We calculate the size of an *UPDATE* message as follows. An *UPDATE* message has to allocate 20 bytes for the IP header, 1 byte for type information, 15 bytes for the IP of the node monitored, 15 bytes for the IP of the message generator, 8 bytes for the *Corrupt* value, and 8 bytes for the *Total* value, and 8 bytes for the timestamp. In total, an *UPDATE* message is 75 bytes long. Figure 9(a) shows the control overhead versus the malice ratio, where $\beta = 25\%$ and group size is 5000. Without linear prediction sampling, each node needs to send out around 17K bytes of data in each submission period, regardless of the malice ratio. This overhead can be significantly reduced by linear prediction sampling. When α is smaller than 10%, the control overhead is cut to less than half. With larger α values, the control

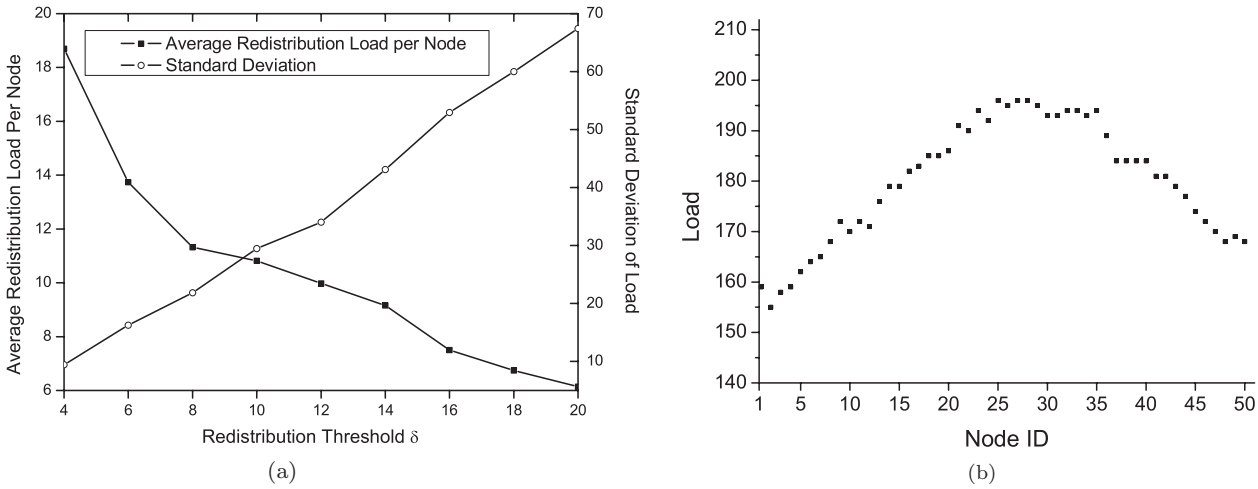


Fig. 10. Performance of dynamic load redistribution among monitoring nodes. (a) Tuning of δ. (b) Load distribution when $\delta = 8$.

overhead keeps increasing. This is because a malicious node sends out corrupt data with a certain probability, and such dynamics lead to more samples taken by their children.

Figure 9(b) shows the overhead for node joining/leaving in an AVL tree. We use the number of messages exchanged between nodes in an operation as the evaluation metric. We evaluate the performance of an AVL tree and a normal binary search tree by varying the number of monitoring nodes (the size of the tree). For both trees, the joining overhead monotonically increases with the tree size. Usually, the larger a tree is, the more deeply a new node will be inserted into the tree. Although an AVL tree requires more subtree adjustment when a node joins, its amortized joining overhead is lower than that in a binary search tree. This is because an AVL tree has a bounded tree height, which further bounds the worst joining overhead. The general trends for node-leaving are similar to node joining. An AVL tree can achieve slightly better performance than a binary search tree. In summary, an AVL tree can achieve low control overhead for node-joining and leaving as compared to a normal binary search tree. Also note that the balance property of an AVL tree can significantly reduce the cost of the much more frequent query and update operations. It is hence more efficient than a normal binary search tree.

Figure 10 shows the performance of the load redistribution mechanism. We set the number of streaming nodes and the number of monitoring nodes to 9000 and 50, respectively. The IP addresses of streaming nodes are uniformly distributed within a given range. Figure 10(a) shows the performance with different redistribution thresholds δ. We define *redistribution load* of a monitoring node as the number of IPs moved to the node from its neighbors or moved to its neighbors from the node. As shown in Figure 10(a), the average redistribution load per node decreases with δ. Clearly, when δ becomes larger, nodes will redistribute their loads less frequently. The redistribution load is hence lower, and the load distribution among nodes becomes more uneven. Therefore, the standard deviation of loads on the monitoring nodes increases with δ.

Figure 10(b) shows the load distribution on the monitoring nodes when $\delta = 8$. We sort the monitoring nodes in ascending order according to their ranges. They are assigned node IDs in that order. As shown in the figure, the maximum load is 26.5% larger than the minimum load. The loads are well balanced among monitoring nodes. Note that the node load first increases and then decreases with the increase of node IDs. This is because the load redistribution is propagated along the tree in a top-down manner. Hence, the root and its close neighbors (with IDs from 20 to 35) have the heaviest loads. After the ranges

have been fully redistributed among all nodes, we believe the load distribution is more related to the IP distribution instead of the distance to the root.

6. CONCLUSION

Many P2P streaming systems assume that nodes cooperate to cache and relay data. However, this may not be true in the open Internet. In this article, we study how to detect malicious nodes in a P2P streaming system. We require each node to keep monitoring its neighbors and to periodically generate monitoring messages. The messages are collected and analyzed at some trustworthy nodes in a monitoring overlay. We study several key components in this framework, including reputation computing in the presence of node lying, reducing monitoring overhead with LP sampling, efficient structure of the monitoring overlay, and load balancing among trustworthy nodes. Our simulation results show that this scheme can efficiently detect malicious nodes with low error, and that our load redistribution method can achieve good load-balancing among trustworthy nodes.

REFERENCES

ABERER, K. AND DESPOTOVIC, Z. 2001. Managing trust in a peer-2-peer information system. In *Proceedings of the ACM Conference on Information and Knowledge Management (CIKM)*. 310–317.

ADAR, E. AND HUBERMAN, B. A. 2000. Free riding on Gnutella. Tech. rep., HP. http://www.hpl.hp.com/research/idl/papers/ gnutella/gnutella.pdf.

BANERJEE, S., BHATTACHARJEE, B., AND KOMMAREDDY, C. 2002. Scalable application layer multicast. In *Proceedings of the ACM Conference of the Special Interest Group on Data Communication (SIGCOMM)*. 205–217.

CHEN, R. AND YEAGER, B. 2001. Poblano: A distributed trust model for peer-to-peer networks. Tech. rep. SUN Microsystems.

CHU, Y. H., RAO, S., SESHAN, S., AND ZHANG, H. 2002. A case for end system multicast. *IEEE J. Sel. Areas Commun. 20,* 8, 1456–1471.

CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. 2001. *Introduction to Algorithms*. MIT Press.

CORNELLI, F., DAMIANI, E., VIMERCATI, S., PARABOSCHI, S., AND SAMARATI, P. 2002. Choosing reputable servents in a P2P network. In *Proceedings of the Conference on the World Wide Web (WWW)*. 376–386.

DAMIANI, E., VIMERCATI, S., PARABOSCHI, S., SAMARATI, P., AND VIOLANTE, F. 2002. A reputation-based approach for choosing reliable resources in peer-to-peer networks. In *Proceedings of the ACM Conference on Computer and Communication Security (CCS)*. 207–216.

DEERING, S. E. 1988. Multicast routing in internetworks and extended LANs. *ACM SIGCOMM CCR '18,* 4, 55–64.

DO, T., HUA, K. A., AND TANTAOUI, M. 2004. P2VoD: Providing fault tolerant video-on-demand streaming in peer-to-peer environment. In *Proceedings of the IEEE International Communications Conference (ICC)*. 1467–1472.

DRAGOVIC, B., KOTSOVINOS, E., HAND, S., AND PIETZUCH, P. 2003. XenoTrust: Event-based distributed trust management. In *Proceedings of the International Conference on Database and Expert Systems (DEXA)*.

GUO, Y., SUH, K., KUROSE, J., AND TOWSLEY, D. 2003. P2Cast: Peer-to-peer patching scheme for VoD service. In *Proceedings of the Conference on the World Wide Web (WWW)*. 301–309.

HABIB, A. AND CHUANG, J. 2006. Service differentiated peer selection: An incentive mechanism for peer-to-peer media streaming. *IEEE Trans. Multimedia 8,* 3, 610–621.

HEI, X., LIANG, C., LIANG, J., LIU, Y., AND ROSS, K. W. 2007. A measurement study of a large-scale P2P IPTV system. *IEEE Trans. Multimedia 9,* 8, 1672–1687.

HERNANDEZ, E. A., CHIDESTER, M. C., AND GEORGE, A. D. 2001. Adaptive sampling for network management. *J. Netw. Syst. Manage. 9,* 4.

JANNOTTI, J., GIFFORD, D. K., JOHNSON, K. L., KAASHOEK, M. F., AND O'TOOLE, J. W. 2000. Overcast: Reliable multicasting with an overlay network. In *Proceedings of the USINEX Symposium on Operating Systems Design and Implementation (OSDI)*. 197–212.

JIN, X., CHAN, S.-H. G., YIU, W.-P. K., XIONG, Y., AND ZHANG, Q. 2006a. Detecting malicious hosts in the presence of lying hosts in peer-to-peer streaming. In *Proceedings of the IEEE International Conference on Multimedia and Expo (ICME)*. 1537–1540.

JIN, X., XIA, Q., AND CHAN, S.-H. G. 2006b. Building a monitoring overlay for peer-to-peer streaming. In *Proceedings of the IEEE Global Telecommunications Conference (GLOBECOM)*.

JUN, S., AHAMAD, M., AND XU, J. 2005. Robust information dissemination in uncooperative environments. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*. 293–302.

KAMVAR, S. D., SCHLOSSER, M. T., AND GARCIA-MOLINA, H. 2003. The EigenTrust algorithm for reputation management in P2P networks. In *Proceedings of the Conference on the World Wide Web (WWW)*. 640–651.

KAZAA. http://www.kazaa.com.

KNUTH, D. E. 1998. *The Art of Programming, Volume 3: Sorting and Searching 2nd Ed.* Addison-Wesley.

LAI, K., FELDMAN, M., CHUANG, J., AND STOICA, I. 2003. Incentives for cooperation in peer-to-peer networks. In *Proceedings of the Workshop on Economics of Peer-to-Peer Systems (P2PECON)*.

LIU, J. AND ZHOU, M. 2006. Tree-assisted gossiping for overlay video distribution. *Kluwer Multimedia Tools Appl. 29,* 3, 211–232.

MARTI, S. AND GARCIA-MOLINA, H. 2006. Taxonomy of trust: Categorizing P2P reputation systems. *Comput. Netw. 50,* 40, 472–484.

MEKOUAR, L., IRAQI, Y., AND BOUTABA, R. 2006. Peer-to-peer's most wanted: malicious peers. *Comput. Netw. 50,* 4, 545–562.

NIELSON, S., CROSBY, S., AND WALLACH, D. 2005. A taxonomy of rational attacks. In *Proceedings of the International Workshop on Peer to Peer Systems (IPTPS)*.

RODRIGUES, R. AND LISKOV, B. 2005. High availability in DHTs: Erasure coding vs. replication. In *Proceedings of the International Workshop on Peer to Peer Systems (IPTPS)*.

ROWSTRON, A. AND DRUSCHEL, P. 2001. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. 188–201.

SHERWOOD, R., LEE, S., AND BHATTACHARJEE, B. 2006. Cooperative peer groups in NICE. *Comput. Netw. 50,* 4, 523–544.

SINGH, A., CASTRO, M., DRUSCHEL, P., AND ROWSTRON, A. 2004. Defending against Eclipse attacks on overlay networks. In *Proceedings of the ACM Special Interest Group on Operating Systems European Workshops (SIGOPS EW)*.

SINGH, A. AND LIU, L. 2003. TrustMe: Anonymous management of trust relationships in decentralized P2P systems. In *Proceedings of the IEEE Conference on Peer to Peer Computing (P2P)*. 142–149.

SSL. Introduction to SSL, http://docs.sun.com/source/816-6156-10/contents.htm.

STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. 2001. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of the ACM Special Interest Group on Data Communications (SIGCOMM)*. 149–160.

TAN, G. AND JARVIS, S. A. 2006. A payment-based incentive and service differentiation mechanism for peer-to-peer streaming broadcast. In *Proceedings of the IEEE International Workshop on Quality of Service (IWQoS)*. 41–50.

TANG, Y., LUO, J.-G., ZHANG, Q., ZHANG, M., AND YANG, S.-Q. 2007. Deploying P2P networks for large-scale live video-streaming service. *IEEE Comm. Mag. 45,* 6, 100–106.

TRAN, D. A., HUA, K. A., AND DO, T. T. 2004. A peer-to-peer architecture for media streaming. *IEEE J. Sel. Areas Commun. 22,* 1, 121–133.

XIONG, L. AND LIU, L. 2004. PeerTrust: Supporting reputation-based trust for peer-to-peer electronic communities. *IEEE Trans. Know. Data Engin. 16,* 7, 843–857.

ZHANG, B., JAMIN, S., AND ZHANG, L. 2002. Host multicast: A framework for delivering multicast to end users. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*. 1366–1375.

ZHANG, X., LIU, J., LI, B., AND YUM, T.-S. P. 2005. CoolStreaming/DONet: A data-driven overlay network for peer-to-peer live media streaming. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*. 2102–2111.

Received September 2008; revised January 2009; accepted January 2009