Fault-Tolerant Data Delivery for Multicast Overlay Networks

Vasileios Pappas
Computer Science Department
University of California, Los Angeles
vpappas@cs.ucla.edu

Beichuan Zhang
Computer Science Department
University of California, Los Angeles
bzhang@cs.ucla.edu

Andreas Terzis
Computer Science Department
Johns Hopkins University
terzis@cs.jhu.edu

Lixia Zhang
Computer Science Department
University of California, Los Angeles
lixia@cs.ucla.edu

Abstract

Overlay networks represent an emerging technology for rapid deployment of novel network services and applications. However, since public overlay networks are built out of loosely coupled end-hosts, individual nodes are less trustworthy than Internet routers in carrying out the data forwarding function. In this paper we describe a set of mechanisms designed to detect and repair errors in the data stream. Utilizing the highly redundant connectivity in overlay networks, our design splits each data stream to multiple sub-streams which are delivered over disjoint paths. Each sub-stream carries additional information that enables receivers to detect damaged or lost packets. Furthermore, each node can verify the validity of data by periodically exchanging Bloom filters, the digests of recently received packets, with other nodes in the overlay. We have evaluated our design through both simulations and experiments over a network testbed. The results show that most nodes can effectively detect corrupted data streams even in the presence of multiple tampering nodes.

1. Introduction

In recent years a number of overlay networks have been developed. These systems aim to provide functionalities that cannot be easily or quickly provided by the IP network layer. For example, overlay networks have been proposed to provide ubiquitous multicast connectivity to end users [8, 24, 6, 2], to provide robust end-to-end connectivity in the face of network outages [1], or to provide protection against DoS attacks [13].

Although overlay networks can provide new functionality and/or better performance to applications, their loose-coupled nature raises new research challenges. One of the primary concerns in using an overlay network is the trustworthiness of its data delivery. Since individual overlay nodes may be owned by anonymous users, data delivery over such a system is potentially exposed to a great number of faults, ranging from innocent data errors to intentional modifications. It is difficult to detect when these faults happen and perhaps even more challenging to repair, because there is no central management or control to oversee the operations in such highly distributed systems.

Well-known fault-detection methods, such as protocols that can achieve Byzantine fault tolerance [14], are not applicable to such large scale distributed systems for two reasons: the number of participants is large and thus protocol complexity is likely to be prohibitively high; in addition fault detection in packet delivery needs to be performed at the packet level, which implies a high message overhead. Furthermore, given the diverse nature of applications, not all applications require the strong fault detection guarantees and the associated high cost provided by Byzantine robustness. For example certain multimedia applications are capable of tolerating a small percentage of data losses.

In this paper we aim to provide a general framework for detecting data stream inconsistencies in multicast overlay networks. Although our description focuses on the application of this framework to shared-tree and source-tree end-host multicast systems, we believe the framework can be easily extended to other overlay services. The proposed methods are lightweight and are specifically tailored for detecting data stream deviations for any application that needs to disseminate data to large numbers of participants.

Our framework includes two methods that are used to detect stream inconsistencies in multicast data delivery, and one method that can be used to identify the exact location of the faults. The first fault-detection method use of Bloom fil-

ters [3], through which each node can compare its received data stream with that of other nodes. The second method takes advantage of the ability to easily construct vertex disjoint paths in overlay networks, and sends additional data integrity checking information through different paths, enabling nodes to verify the validity of the received packets.

The main contributions of this work are the following:

- A set of general techniques that can detect a set of faults that may appear during the packet forwarding process in an overlay network.
- A method for identifying the exact point of failure and a method for isolating the faulty parts.
- An evaluation of the proposed schemes through both simulation and a prototype implementation.

The rest of the paper is organized as follows: in section 2 we present the kind of faults that may occur in an overlay network. In sections 3 and 4 we describe how our framework works. In section 5 we evaluate our techniques through simulation and a prototype implementation. Section 6 discusses related work and Section 7 summarizes our work.

2. Fault Types and Causes

In an overlay network environment, where end users collaborate in the packet forwarding process, various kinds of faults may occur, ranging from routing faults due to incorrect forwarding information to data stream faults where undetected data alterations occur during forwarding. The causes of faults may include, but not limited to, the following:

- **Software bugs**: A software bug in one node can cause problems that may affect the entire system. Up to now most, if not all, of the overlay system designs assume a fail-stop model for participating nodes. However practice shows that there are faults that do not follow that model.
- **Malicious code**: Given that any user can install arbitrary code on his own node as long as it appears to conform to the overlay protocol at a minimum level, it is relatively easy for a malicious user to disrupt the correct execution of the protocol and cause problems that affect other users. In addition, it is also possible to have a set of malicious nodes that collude to make fault detection more difficult.

3. Fault Detection

The two fault detection methods, described later in the paper, are based on the following two basic principles used by the majority of fault-tolerant systems:

- **Comparison**: two or more modules that perform the same operation compare their results. If there is disagreement, an error has been detected. Methods such as majority vote can then be used to pinpoint the faulty modules.
- **Self-checking**: a single module can validate the result of its own operation by carrying out an additional check using redundant information. A simple example of this method is the detection of corrupted information with the use of error detection codes.

3.1. Comparison Method

This method takes advantage of the fact that under ideal conditions all the nodes that belong to the same multicast group, have the same view of the data stream. Two or more nodes have the same view of the stream if they receive the same packets within a bounded time difference, not necessarily in the same order. Thus every node can check whether its received data stream diverges from that of other nodes.

A stream deviation is detected if during the comparison phase there is an indication of a missing packet, a modified packet, or an additional packet. Although the comparison method is conceptually simple, applying it to overlay data delivery imposes a great challenge. The number of nodes in an overlay network can be large, which prohibits packet comparisons among all the nodes. Furthermore, the amount of data to be compared can be immense.

3.1.1. Stream Comparison with Bloom Filters A naive solution of packet-to-packet comparison may increase the overall network traffic at least *t* times, where *t* is the number of comparisons required for each packet. A better approach is to use collision-resistant hash functions which can produce constant size values for each packet, and to perform the comparison using the hash values. As an example the MD5 hash function [19] can be used to reduce the size of the transmitted information: instead of sending the whole packet a node can send the 128-bit message digest of each packet. This method can considerably reduce the overall message overhead compared to the packet-to-packet approach, given that most multicast applications, such as video and audio streaming, tend to use large size data packets. Even if the average message size is 512 bytes (most packets are closer to the Ethernet MTU value of 1500 bytes), the gain is 96.875% savings of network traffic load.

In order to reduce the traffic overhead even further, we encode the message digests of the data stream by using Bloom filters [3]. A Bloom filter is a space efficient data structure that supports membership queries and consists of an array of *m* bits and a set of *k* independent hash functions, H_1,H_2,...H_k, with range $[0, m-1]$. Given that we want to store a message A in that structure, we com-

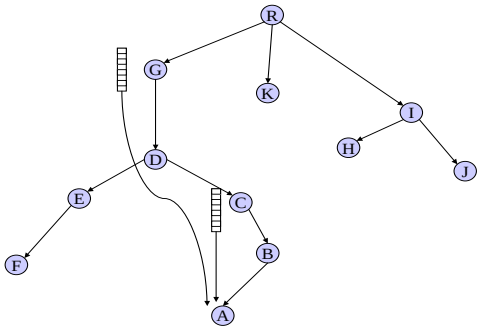


Figure 1. Comparison method with Bloom filters. Node A receives the Bloom filter from the upstream nodes C and G.

pute the hash values $H_1(A)$,$H_2(A)$,...$H_k(A)$ of the message and we set to 1 the bits of the Bloom filter that correspond to these hash values. In order to check if a message X is stored in the Bloom filter we check whether the $H_1(X)$,$H_2(X)$,...$H_k(X)$ bits of the Bloom filter are all set to 1. If not, it is clear that the message is not stored in the filter, otherwise there is a probability that this message has been stored. That probability depends on the size *m* of the Bloom filter, the number *k* of hash functions and the number *n* of the messages that have been stored. It has been proved [11] that the probability of a false positive is: $f = (1 - e^{-kn/m})^k$. Bloom filters have recently been used in a variety of networking applications [11, 22, 10, 4], mainly because they have the attractive property of considerably reducing the size of transmitted information, by trading off the probability of false positives. By using Bloom filters, we are able to reduce even more the overhead of the comparison method, given that under specific parameter settings, Bloom filters require around 10 times less traffic compared to the MD5 hash approach.

Each node maintains two data structures: a list of MD5 hash values of the recently received packets and a Bloom filter that stores these values. The maximum number n of packets stored in the Bloom filter depends on the percentage of false positives that the system can sustain and it can be derived from the above equation for a given number of hash functions k and a given size m. The Bloom filter is the only information that is exchanged between two nodes that want to perform a stream comparison. The filters are sent over a direct IP connection (bypassing the overlay network and thus not allowing malicious nodes to modify the filters). Additionally, we use collision resistant hash functions H_i to prevent malicious nodes from easily selecting modified packets that have the same hash value to a legitimate packet (the probability of successful tampering in this case is equal to the false positive probability).

When a node A receives a Bloom filter from another node C, the former checks whether each MD5 hash value, stored in the list of recently received packets, appears in C's Bloom filter. If not, a stream deviation has been detected that is due to packet addition, deletion or modification, depending on the relative position of nodes A and C and the forwarding direction of the disputed packet. If the packet is forwarded with a direction from A to C then it has been deleted or been modified, otherwise it has been added somewhere between A and C. We must mention that a direct comparison between the two Bloom filters, such as a bit-to-bit comparison, can not indicate the exact number of missing packets given that more than one packets can possibly set the same bit.

There are two methods to maintain the probability of false positives in the Bloom filter for the recently received packets below a desired threshold. We either reinitialize the Bloom filter periodically or we remove the oldest packet for every new packet that is inserted in the filter, when the filter reaches its maximum capacity. The first solution is relatively simple, but it assumes either that the nodes are synchronized and exchange the filters just before they reset them, or that they exchange a number of previous filters in addition to the current one. The second approach works if we associate with each bit of the Bloom filter a reference counter. Whenever a packet is inserted in the Bloom filter the counters of the bits, that is pointing to, are increased by one, and whenever it is removed they are decreased. If the counter is higher that 0 then the corresponding bit of the filter is 1 otherwise is zero. The second technique shares many similarities with the one used in [11]. For simplicity, we have chosen the first approach, where the last two filters are exchanged and no synchronization is required.

3.1.2. Node Selection The detection power of the comparison method depends on the nodes selected for the Bloom filters exchange. The most suitable nodes are the data stream sources, but that solution does not scale neither with the number of sources nor with the total number of overlay nodes. On the other hand, the solution of randomly picking a node is not effective because it is expected to give a considerable number of false positives, in the cases where a node receives the correct stream but it assumes that there is an error, because the majority of the nodes that performs the comparison with, happen to receive a diverged stream. In section 5 we will show that in most cases the solution of randomly picked nodes performs poorly compared with the following two proposed schemes.

In the simple case where there is only one malicious or faulty intermediate node M, there is at least one node A that can identify any stream deviations, due to the faulty

node M. Node A can be any node directly connected to M, for which node M forwards packets, and it can identify the stream deviation if it compares its stream with all the other nodes directly connected to M. If there is a packet mismatch between A and any other neighboring node N of M, then there is an indication that either M or N modified the packet. Thus a node needs to compare its stream only with all the nodes that are two overlay hops away. In that case if the comparison method indicates a deviation in the stream then either the first or the second hop node is responsible for modifying the data. We must note that, in the case that M is malicious or follows an arbitrary failure model, it is pointless for node A to do the comparison with M given that it may show no stream deviation, even if there is. Moreover node A needs to perform the comparison only with the nodes that are in the direction from which it receives a data stream.

In order to handle the cases where many colluding nodes are connected one after the other on the same path, the comparison needs to be performed with some additional nodes $n = 1, 2, 3..$ hops away towards the root of the tree. A potential problem of this approach is that nodes that are closer to the root get overloaded, given that they receive a total number of bloom filter queries that increases exponentially with the tree depth. For this reason each node asks the bloom filters from nodes that are $n = 2^k (k = 1, 2, 3...)$ hops away. Assuming a binary tree, the nodes that are closer to the root of the tree receive a number of queries $Q = (4^{floor(log_2(L-l-1))} - 1)/3 + 1$, instead of $Q = 2^{L-l} - 2$, where L is the tree depth and l is the distance to the root.

3.2. Self-Checking Method

The intuition behind the self-checking method is to transmit some additional information related with one packet, which can be the checksums or a hash value of the packet payload, and to verify the integrity of the packet based on that information. The checksum information needs to be protected from any kind of modification and one way to achieve this is by the use of digital signatures. While there is work on the authenticity and integrity of multicast data stream [16, 23] we decided to follow a slightly different approach. The main reason is that these solutions either require some additional infrastructure, such as PKI or time synchronization services, which makes the whole system design more complex and dependable on external factors, or they require a considerable message and computational overhead.

3.2.1. Splitting the Stream over Multiple Paths Our approach is based on the fact in an overlay network it is relatively easy to create multiple disjoint paths [1]between a source and a receiver node. In section 3.2.2, we describe a simple algorithm for the creation of multiple paths in shared trees, but in general the multiple disjoint path creation problem can be solved using the maximum flow problem [9]. A faulty or malicious node located on one path can alter the packets traveling through that path but it cannot modify the packets that are traveling through different paths. Thus if we send the checksum information, used for verification, through a different path, a malicious node can change only the data packet or the packet that contains the checksum, but not both of them.

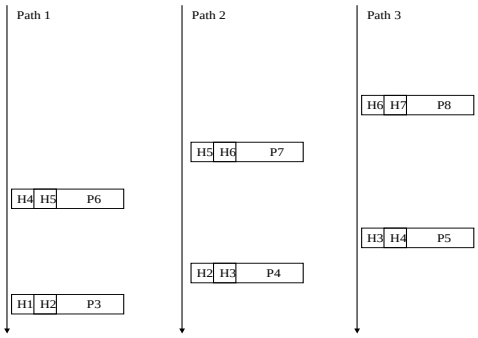


Figure 2. Self-Checking Method with 3 paths and 2 hash functions. Packet number 5 (P5) carries in addition the hash values of packet 3 (H3) and packet 4 (H4), and uses a different path.

The same method can be extended in order to ensure the message integrity under the presence of colluding nodes. A set of different checksum information, which refer to the same packets and which are able to verify the integrity of the packet independently from each other, are piggybacked to subsequent packets and transmitted through different paths. Thus if the majority of the checksums verify the message, the message is considered authentic. It is clear that if a node uses *t* different paths, then this method is guaranteed to work correctly only if the paths are non overlapping and if more than the half of the paths are free from malicious nodes that are in collusion.

More specifically, this method works as follows: for each packet P the source node computes *k* hash values by using k independent hash functions, H_1,H_2,...H_k. Each of these values is appended to *k* subsequent packets, thus every packet carries k hash values, one for each of the *k* previous data packets. In addition, each receiver must establish *t* different paths. The number of paths *t* must be greater than the number of the hash functions in order for a packet and its *k* hash values to follow different paths.

1 Disjoint paths refer to overlay network paths and not to physical layer paths

```
void FindDisjointPaths(RP,GroupID) {
  // I: list of invalid nodes
  // S: list of nodes belonging to other paths
  // P: stack of potential parents
  I,S, P = {};

  RootList = FindRoots(RP,GroupID);
  for (R in RootList) {
    PotentialParent = R;
    FoundParent = false;

    while ( ! FoundParent ) {
      ChildrenList = FindChildren(PotentialParent);
      while (ChildrenList not empty) {
        NearestNode = FindMinRTT(ChildrenList, PotentialParent);

        if (NearestNode not in S && NearestNode not in I) {
          if (NearestNode != PotentialParent) {
            P.push(PotentialParent);
            S.add(PotentialParent);
            PotentialParent = NearestNode;
          } else {
            accept = SendJoinRequest(PotentialParent);
            if (!accept) {
              I.add(PotentialParent);
              PotentialParent = P.pop();
           } else {
              FoundParent = true;
              S.add(PotentialParent);
              P = {};
            }
          }
          break;
        } else {
          ChildrenList.remove(NearestNode);
          if (ChildrenList is empty) {
            backtrack = true;
          }
        }
      }

      if (backtrack) {
        PotentialParent = P.pop();
      }
    }
  }
}
```

Figure 3. Pseudo-code for the creation of multiple disjoint paths

Different packets traverse different paths based on a sequence identifier. The identifier is set by the source of the stream, and a certain path carries only the packets whose sequence identifiers is equal to the path identifier modulo the total number of paths *t*. Thus the stream is split in multiple sub-streams, and packets traverse the different paths in a round robin fashion. While this approach can cause reordering of packets, this can be handled at the transport or application layer.

The verification process is performed on every node by checking if the hash values, received from the subsequent packets, match the k locally computed hash values for the original packet. If the majority of the hash values matches with the packet contents the packet is considered unmodified. Figure 2 shows a sequence of packets with two hash functions and 3 different paths.

3.2.2. Multiple Paths Construction Figure 3 gives the pseudo-code for the creation of multiple disjoint paths for shared tree multicast overlays. It is used by every node whenever it joins the overlay network and whenever there is a change in the tree structure. Briefly the node that runs this algorithm starts querying the root and the nodes below it in a depth first manner, in order to find the closest possible node. Then, it establishes the first sub-path with this node. Similarly it repeats the same procedure but without querying the nodes that have already been selected on any of the established sub-paths. At the end it will establish *t* disjoint paths. We must mention that in the case that this establishment is impossible (when the number of participants is small), it is allowable to reuse some of the already selected nodes, in which case the total number of disjoint paths is reduced.

4. Fault Repair

Whenever a faulty or malicious node is detected the non-faulty nodes need to isolate the problematic node, by removing it out of any forwarding path. Thus, fault repair is a two step process: first, when a node detects a deviation in the data stream, it initiates a procedure to pinpoint the faulty node, and second, in collaboration with the other nodes puts the faulty node aside.

The identification of the faulty node is done by querying a number of upstream nodes along the path of the deviated stream, which are $n = 2^k (k = 1, 2, 3...)$ hops away. The query takes the form of a request for the Bloom filter for the recently received packets. With those filters a node is able to check which nodes received the same faulty packets, and thus to estimate its distance to the faulty node. That distance is expressed as number of overlay hops. In the case that the comparison method is used for the fault detection, then the querying step can be omitted, given that the Bloom filters are already known. On the other hand, the self-checking method can detect the faults almost instantly. Thus, a combination of these two methods can guarantee a fast detection of a fault and an exact identification of the faulty node.

A node that has already estimated the distance to the faulty one, defers to take any action to isolate that node, for an amount of time that is proportional to its distance from the faulty node, in the hope that nodes closer to the faulty one will isolate it. Thus the nodes that are directly connected to the faulty node will be the first that will isolate it. In the case of shared-tree protocols, the isolation is performed by switching from the erroneous parent to the node that is the closest after the faulty one, and in the case of source tree protocols the isolation is done by artificially increasing the proximity metric to the faulty node. The new proximity takes its maximum possible value, and thus the faulty node cannot be on the shortest path to the source. In addition, the nodes, which are in a pending state of changing parent, cancel their repair procedure when they detect a change on the path to the root , given that this change indi-

cates that an ascendant node took care of isolating the faulty node.

For nodes that have been identified as faulty in the past, it's preferable not to continue considering them faulty for an infinite time, since the identification procedure is not always accurate and because faulty nodes might actually be repaired after some time. Thus nodes that have appeared as faulty many times in the past, can be considered as faulty with higher confidence. We make use of a quarantine list in order to handle the above cases. When a node pinpoints another node as a faulty one, it inserts it to its quarantine list. That list is maintained on every node and stores all the nodes that have recently been identified as faulty. A faulty node stays in that list for an amount of time that increases exponentially with the number of times that this node has been identified as faulty.

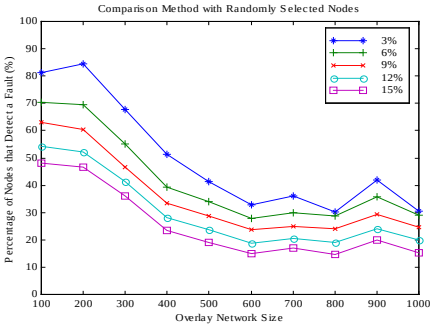
5. Evaluation

We have evaluated the fault detection power of both the comparison and self-checking methods by simulation, and tested a prototype implementation over the Internet.

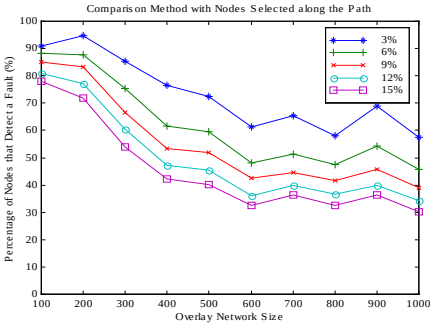
5.1. Simulation

We use two metrics to measure each method's fault detection power. The first metric is the percentage of nodes that can detect a fault in a given network with some malicious nodes. The other is the rate of false positives. Three methods are compared in the simulation: the comparison method with randomly selected nodes, the comparison method in which nodes to be inquired are on the path to the source, and the self-checking method. In this section we present the fault detection results while in the following section we show the combined results of fault detection and repair.

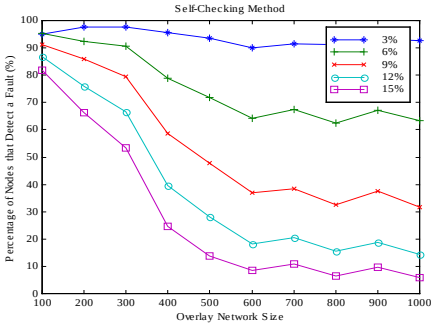
An important factor in evaluating a fault detection method is the attack model of malicious nodes. In order to get the lower bound of our methods' performance, we made two strong assumptions on the attack model. First, we assume all malicious nodes are in full collusion. Each malicious node modifies not only data payload, but also the checking information carried by subsequent packets to match the modified payload. Different malicious nodes will make consistent modifications, so that checking information forwarded by one matches the data payload forwarded by another. Second, we assume the number of malicious nodes increases linearly with the network size, that is, their percentage keeps unchanged as the network grows. In reality, it is more likely that a large overlay network has a very small number of malicious nodes, of which only a few collude with each other. Therefore, we expect our methods generally perform bet-



(a) Comparison Method: Randomly selected nodes

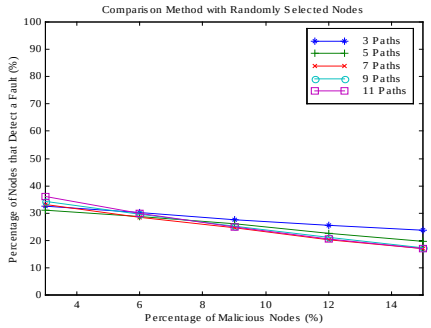


(b) Comparison Method: Selected nodes belong on the path to the root

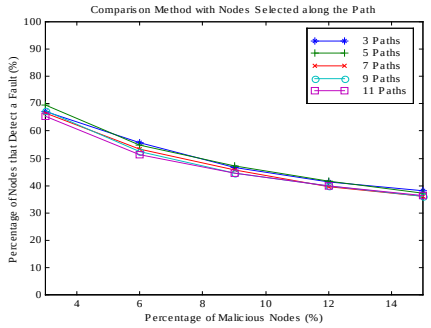


(c) Self-Checking Method

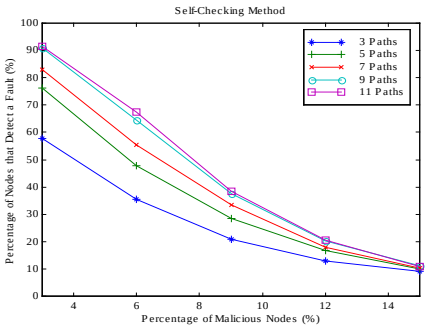
Figure 4. Percentage of nodes that can detect a fault in a network of 11 disjoint paths



(a) Comparison Method: Randomly selected nodes



(b) Comparison Method: Selected nodes belong on the path to the root



(c) Self-Checking Method

Figure 5. Percentage of nodes that can detect a fault in a network of 700 nodes

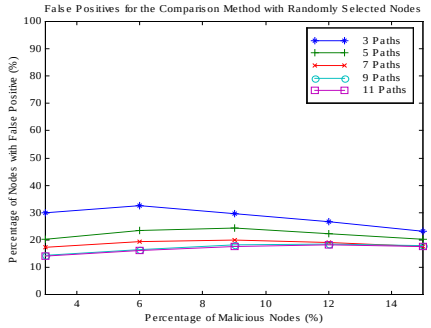
ter in the Internet than the results shown below. Performance evaluation under other attack models is subject of future work.

The simulation is done on an Internet-like topology generated by Inet [12]. Out of the 10,000 nodes in the topology, 200 to 1000 nodes are randomly selected and used to build an overlay network by HMTP [24]. We randomly pick a source and a set of malicious nodes in each run. The results presented below is the average over 1000 runs with different sources and malicious nodes.

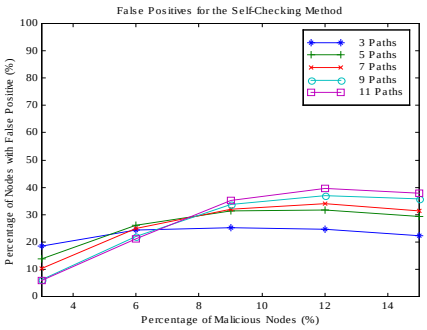
Figures 4(a), 4(b) and 4(c) show the percentage of nodes that can detect a fault under different network sizes, ranging from 100 to 1000 nodes. The percentage of malicious nodes ranges from 3-15% of the network size and the number of disjoint paths is fixed at 11. Note, in each curve, the number of malicious nodes increases linearly as the network size increases since the percentage is fixed. It is evident that choosing nodes on the path to the source performs better than randomly choosing nodes to inquire in the comparison method. From these figures we can also infer each method's working range under our attack model: For small network sizes, both methods can detect faults with high probability even if the percentage of malicious node is quite high; For larger network sizes, the comparison method performs better than the self-checking method except when the percentage of malicious nodes is 3%. When the network grows, the overlay path length increases since the HMTP overlay maintains a maximum node degree.This path length increase, coupled with the growing number of colluding malicious nodes, gives higher probability of having at least one malicious node on the path to the source in large networks. This is why the detection power decreases for both methods as the network increases. One possible way to overcome this limitation is to increase the connectivity degree of each node to achieve shorter path length in large overlay networks.

In the self-checking method, one malicious node on a path is enough to compromise the data sub-stream to all downstream nodes. But in the comparison method, it takes multiple malicious nodes to fool a downstream node. Therefore, when the number of disjoint paths is fixed, the self-checking method is sensitive to the number of malicious nodes, while the comparison method is only sensitive to the percentage of malicious nodes. This is why self-checking method does not perform as well as comparison method when the number of colluding malicious nodes is large, i.e. large network size and high percentage of malicious nodes.

Figures 5(a), 5(b) and 5(c) show detection power of these methods in a network of 700 overlay nodes with different number of disjoint paths. It is clear that the number of disjoint paths can affect the detection power of the self-checking method: when the number of paths increases, the number of nodes that can detect a fault increases too. This is



(a) Comparison Method



(b) Self-Checking Method

Figure 6. False positives rate in a network of 700 nodes

expected because the possibility of having a malicious node on each path decreases when the number of disjoint paths increases. On the other hand, the detection power of the comparison method is almost independent from the number of disjoint paths because the comparison is performed with nodes belonging to the same path each time.

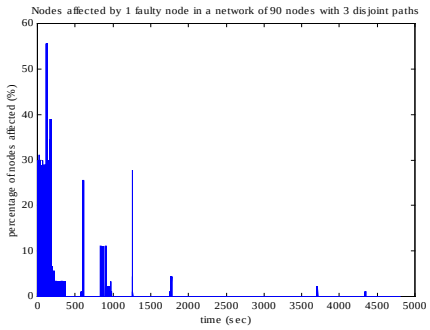
Figures 6(a) and 6(b) show the false positive rate of the comparison and the self-checking methods in a network of 700 nodes with the number of malicious nodes ranging from 3% to 15%. In the self-checking method, when the percentage of malicious nodes is low, most paths are free of malicious nodes. Therefore the increase in the number of paths reduces the false positive rate. However, after a certain point, when the percentage of malicious nodes is high, paths with malicious nodes become dominant over paths free of malicious nodes. The increase of such "bad" paths gives higher chance for false positive. In the comparison method, it takes many malicious nodes to totally compromise a path. Therefore, in our range of malicious nodes, the usable paths are always dominant, and increasing them gives lower false positive rate.

5.2. Implementation

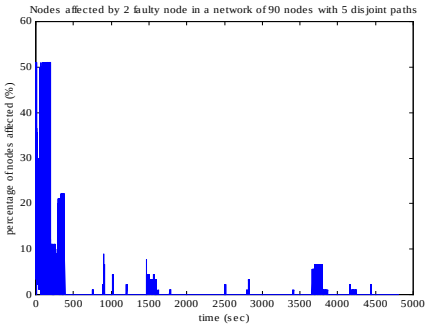
We implemented and tested a prototype in order to evaluate these techniques on a real overlay network deployed in the Internet. The implementation is based on the HMTP [24, 25] protocol, runs at the application level, and has been developed for different flavors of the Unix operating system. It consists of a daemon process, which can run either as a privileged or user process and a set of library functions that third-party applications can use in order to communicate with the daemon process. The daemon participates in the overlay network and forwards packets on behalf of the applications. The prototype implements both the self-checking and the comparison method, and uses a combination of them in order to detect the faulty nodes. More specifically, the comparison method is triggered when the Bloom filter of the daemon is reset. In addition the self-checking method is able to trigger the comparison method whenever it detects a fault over a certain path. Fault repair is also implemented.

We have deployed this prototype system over the Internet by using the PlanetLab testbed [18]. The testbed consists of about 100 machines located at around 40 different sites. The total number of nodes in each experiment is 90. We insert a number of malicious nodes that alter the content of each packet. The faulty nodes are in collusion and are intentionally placed close to the source of the stream in order to initially affect a large number of nodes. Each experiment runs for 90 minutes.

Figures 7(a) and 7(b) show the percentage of nodes that receive faulty packets during the 90 minutes, with 3 and 5 disjoint paths, and 1 and 2 malicious nodes, respectively. At the beginning of the experiment, a large number of nodes receive faulty packets since the malicious nodes are close to the source. But as time goes by, nodes are able to detect faults, pinpoint the malicious node, and exclude them from the forwarding path. Therefore, the peaks in both Figure 7(a) and 7(b) diminish by the time of 500 seconds. Spikes after 500 seconds are due to two reasons: either nodes that were able to identify the malicious nodes in the past start using them again because their entry in the quarantine list has expired, or new nodes that have never identified the malicious nodes connect to them for the first time. It is clear that over a time period, our techniques are effective in detecting, locating and repairing faults in real Internet experiments. The time needed to identify a malicious node is proportional to the number of faulty application data packets, and thus it is proportional to the applications' average transmission rate.



(a) 3 disjoint paths, 1 faulty node



(b) 5 disjoint paths, 2 faulty nodes

Figure 7. Percentage of nodes that get affected by faulty nodes in a network of 9 0 nodes

Finally we use the prototype implementation to estimate the processing overhead of the comparison and self-checking method. For these experiments we used two Intel P4 3GHz machines running Linux 2.4.18 connected by a 100Mbps Ethernet link. Packets are forwarded from one overlay node to the other at different rates and with different packet sizes. Table 1 shows the increase in the total forwarding time that is due to the above two techniques. These numbers represent the percentage of time spent on these methods over the total time spent in packet forwarding. Even if the overhead of computing the MD5 hash values of packets is an increasing function of the packet size, the overhead introduced by these methods is almost the same for different packet sizes under the same transmission rate, because the time for transmitting and receiving a packet is still an increasing function of the packet size. The reception and transmission overhead introduced in the application is due to the memory copies between kernel and user

Transmission Rate	8 bytes	128 bytes	512 bytes	1024 bytes
1 pkt/sec	16.219	16.670	17.185	18.271
10 pkts/sec	19.053	19.297	20.031	20.638
100 pkts/sec	22.249	23.769	23.900	23.982
1000 pkts/sec	21.681	22.160	23.069	27.113

Table 1. The processing overhead (%) of the comparison and self-checking method introduced in the forwarding process of each packet (the size corresponds to the payload size)

space, and increases linearly with the packet size. The message overhead of these two methods greatly depends on the specific parameters (filter's hash functions, number of disjoint paths) and on the average application packet size. For our settings, where the bloom filter size is 512 bytes and the false positives are 10%, the message overhead of the comparison method is around 2%, whereas for the self-checking method the overhead is 4.7% and 7.8% for 3 and 5 paths respectively, assuming average application data size of 512 bytes.

6. Related Work

The system closest to our work is [5], which studies attacks aimed at preventing correct message delivery in structured peer-to-peer overlay networks and which presents defenses to these attacks. More specifically the authors concentrate on techniques for secure node joining, routing table maintenance and message forwarding. They introduce the secure routing table concept, which is used as alternative routing table whenever a node detects that there is a fault in the message forwarding process. Our techniques can also detect any fault in the message forwarding process and in addition can provide a mechanism for locating the exact node that causes the fault.

A number of methods for fault isolation in multicast trees are also presented at [17], where the authors are mainly interested in faults that are due to lost packets. The authors consider the case of IP multicast networks and make use of the mtrace tool and packet counters in order to estimate the number of lost packets. The two methods proposed in this paper deal with arbitrary types of faults and thus they can be viewed as more generic. Especially the comparison method can be easily used in the case of IP multicast networks, given that it doesn't require the existence of multiple disjoint paths, while the self-checking method cannot directly be applied because it requires major changes in the IP multicast routing protocols.

Extensive research has been conducted in the area of fault-tolerant systems, ranging from work done on file systems [7], to operating systems [20] and Internet services [21]. This paper describes how to apply some well-known high-level concepts of the fault-tolerant systems design in a networking environment and especially in the field of overlay networks. The methods presented at [15] are developed for updating replicas in a Byzantine environment, and are similar in concept with our comparison method. The major difference is that we are interested in locating the exact point of failure, whereas the authors of [15] concentrate in providing hard guarantees for correct replica updates.

The large body of work in the area of secure multicast (e.g. [16, 23]) is related to our work in the sense that a secure system is less vulnerable in a variety of faults. Even if these schemes have been recently optimized for less computational and message overhead, they rely on the existence of services such as PKI or time synchronization services. Our framework can guarantee the integrity of multicast data-streams, without any additional service requirement. Another difference is that while the secure multicast approach is able to detect a fault it cannot pinpoint the exact fault location.

7. Conclusion

In this paper we identified a number of faults that are expected to be a common case in a widely deployed overlay network and we designed a generic framework that can by applied in order to detect and repair that kind of faults. We described how this framework works in the case of shared tree and source tree overlay multicast system and we provided simulation results that support our arguments. Finally we implemented a prototype system that uses the techniques presented in this paper and we verified their feasibility on a real overlay network deployed in a wide testbed over the Internet.

References

[1] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient overlay networks. In *Proc. of the 18th Symposium on Operating Systems Principles*, pages 131–145, Oct. 2001.

[2] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable application layer multicast. In *Proc. of ACM SIGCOMM*, 2002.

[3] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[4] J. Byers, J. Considine, M. Mitzenmacher, and S. Rost. Informed content delivery across adaptive overlay networks. In *Proc. of ACM SIGCOMM*, 2002.

[5] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Secure routing for structured peer-to-peer overlay networks. In *Proc. of the 5th Symposium on Operating Systems Design and Implementation*, 2002.

[6] Y. Chawathe, S. McCanne, and E. Brewer. RMX: Reliable multicast for heterogeneous networks. In *Proc. of IEEE INFOCOM*, Mar. 2000.

[7] P. Chen, E. Lee, G. Gibson, R. Katz, and D. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, 1994.

[8] Y. Chu, S. G. Rao, and H. Zhang. A case for end system multicast. In *Proc. of ACM SIGMETRICS*, pages 1–12, June 2000.

[9] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[10] C. Estan and G. Varghese. New directions in traffic measurement and accounting. In *Proc. of ACM SIGCOMM*, 2002.

[11] L. Fan, P. Cao, J. Almeida, and A. Broder. Summary cache: a scalable wide-area web cache sharing protocol. In *Proc. of ACM SIGCOMM*, pages 254–265, 1998.

[12] C. Jin, Q. Chen, and S. Jamin. Inet: Internet topology generator, 2000.

[13] A. Keromytis, V. Misra, and D. Rubenstein. SOS: Secure overlay services. In *Proc. of ACM SIGCOMM*, Aug. 2002.

[14] Lamport, Shostak, and Pease. The byzantine generals problem. *ACM Trans. Prog. Lang. and Systems*, 4:382–401, 1982.

[15] D. Malkhi, Y. Mansour, and M. Reiter. On diffusing updates in a byzantine environment. In *Symposium on Reliable Distributed Systems*, pages 134–143, 1999.

[16] A. Perrig, R. Canetti, J. Tygar, and D. Song. Efficient authentication and signing of multicast streams over lossy channels. In *IEEE Symposium on Security and Privacy*, pages 56–73, 2000.

[17] A. Reddy, R. Govindan, and D. Estrin. Fault isolation in multicast trees. In *Proc. of ACM SIGCOMM*, pages 29–40, 2000.

[18] I. Research. Planet Lab. http://www.planet-lab.org/, 2002.

[19] R. Rivest. The md5 message-digest algorithm. RFC 1321, IETF, Apr. 1992.

[20] R. Rodrigues, M. Castro, and B. Liskov. BASE: Using abstraction to improve fault tolerance. In *Proc. of the 18th Symposium on Operating System Principles*, pages 15–28, Oct. 2001.

[21] Y. Saito, B. Bershad, and H. Levy. Manageability, availability and performance in porcupine: a highly scalable, cluster-based mail service. In *Proc. of the 17th ACM Symposium on Operating Systems Srinciples*, pages 1–15, 1999.

[22] A. C. Snoeren. Hash-based ip traceback. In *Proc. of ACM SIGCOMM*, pages 3–14, 2001.

[23] Wong and Lam. Digital signatures for flows and multicasts. *IEEE/ACM Transactions on Networking*, 7, 1999.

[24] B. Zhang, S. Jamin, and L. Zhang. Host Multicast: a framework for delivering multicast to end users. In *Proc. of IEEE INFOCOM*, June 2002.

[25] B. Zhang, S. Jamin, and L. Zhang. Universal IP multicast delivery. In *NGC*, Oct. 2002.