FRTR: A Scalable Mechanism to Restore Routing Table Consistency

Lan Wang, Daniel Massey, Keyur Patel, Lixia Zhang

Abstract

This paper presents a scalable mechanism, Fast Routing Table Recovery (FRTR), for detecting and correcting route inconsistencies between neighboring BGP routers. A periodic update approach, used by most routing protocols, is considered infeasible due to the large size of today's global routing table. In FRTR neighboring routers periodically exchange Bloom filter digests of the routing state to detect any potential routing inconsistency. The digests can also facilitate efficient recovery after a BGP session reset. FRTR achieves a low bandwidth overhead by using small digests, and achieves strong consistency by "salting" the digests with random seeds to overcome Bloom filter's false-positive drawback. Our analysis and simulation results show that, with one round of message exchange, FRTR can detect and recover over 91% of random errors that the current BGP would have missed and the overhead can be as low as 1.3% of a full routing table exchange. With salted digests FRTR can detect all the errors with a probability close to 100% after a few rounds of exchanges.

Keywords: *Path-vector routing protocols, Routing resilience, Fault Tolerance, Bloom Filters, System design*

This work is partially supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No DABT63-00-C-1027, by National Science Foundation (NSF) under Contract No ANI-0221453, and by a research grant from Cisco Systems. Any opinions, findings and conclusions or recommendations expressed in this article are those of the authors and do not necessarily reflect the views of the DARPA, NSF, or Cisco Systems.

Lan Wang is with UCLA. Address: UCLA Computer Science Department, 4805 Boelter Hall, Los Angeles, CA 90095. Phone: 310-825-4838. Fax: 310-825-2273. Email: lanw@cs.ucla.edu. She is the contact author.

Daniel Massey is with USC/ISI. Address: 3811 North Fairfax Drive, Suite 200, Arlington, VA 22203. Phone: 703-812-3731 (phone). Fax: 703-812-3712. Email: masseyd@isi.edu

Keyur Patel is with Cisco Systems. Address: 510 McCarthy Blvd, Milpitas, CA 95035. Phone: 408-526-7183. Fax: 408-527-4783. Email: keyupate@cisco.com

Lixia Zhang is with UCLA. Address: UCLA Computer Science Department, 4531G Boelter Hall, Los Angeles, CA 90095. Phone: 310-825-2695. Fax: 310-825-2273. Email: lixia@cs.ucla.edu

I. INTRODUCTION

Dependable and efficient neighbor to neighbor communication is an essential part of distributed routing protocols. To achieve this, intra-domain routing protocols such as RIP [**?**] and OSPF [**?**] use a “soft-state” approach where routers periodically exchange their latest routes or connectivity information. Information that is not refreshed times out and this provides protection against a variety of expected and unexpected failures. Unfortunately this periodic update approach is infeasible for Internet inter-domain routing due to the large size of today’s global routing table. BGP [**?**], the de-facto inter-domain routing protocol, instead uses an event driven update approach and a BGP route never expires unless explicitly removed (i.e. “hard-state”). Once router A has sent its initial routing table to neighbor B, no further route updates are sent unless a route change occurs or the peering session breaks. BGP expects that routing updates could be lost, re-ordered, or corrupted during transmission. To counter these problems, neighboring BGP routers establish a TCP connection and then exchange routing updates over this reliable connection.

Operational experience ([**?**] and [**?**]) has shown that reliable update delivery via TCP alone is inadequate to ensure routing consistency between neighbors. For example, on Oct. 25, 1998, an ISP accidentally originated a large number of invalid routes, creating an outage over large regions of the Internet [**?**]. The faulty AS immediately withdrew the routes. However, some of the withdrawn routes were still present in some areas of the Internet on the following day. Even if we take into account the processing and propagation delay of the messages at every router as well as the slow convergence of BGP [**?**], it should not have taken such a long time to purge the invalid route. The exact cause of this event is unknown, but the results are not surprisingly. Earlier BGP implementations by a few vendors forwarded withdrawal messages to some, but not all the peers in a peer group, causing stale routes to remain in routers that did not receive the withdrawal messages [**?**]. In the BGP “hard-state” approach, these stale routes persist until some external event (such as a peering session reset) flushes the entire routing table.

Routing state can also be altered by attacks [**?**] or corrupted by hardware failures or software bugs. If a peering session is on a shared medium and is not protected by MD5 checksum, another node on the wire can inject a false update. There are also numerous examples of routing bugs and as far back as during the ARPANET operation in 1970’s, a memory fault caused an east coast router to falsely announce a zero cost route to UCLA [**?**]. In the context of BGP neighbor communication, corruption of the Incoming Routing Information Base (RibIn) where all the routes received from a neighbor is kept can cause a router to have an incorrect view of its neighbor’s routes. As a result, when the router recomputes its route to a destination, it may choose a corrupted route from the RibIn (e.g. the route may appear to have the shortest path among all the candidate routes). Overall, the use of reliable communication does not protect against *unexpected* faults.

In addition, transient session failures also occur and BGP lacks an efficient mechanism to synchronize neighboring routers' routing tables after a transient session failure. [**?**] observed that, in the Sprint network, links go down every 30 minutes on average and in 80% of the cases links come back up in less than 10 minutes. Examination of BGP update logs during the Nimda worm attack in September 2001 showed that some BGP monitoring sessions were frequently reset , possibly due to the congestion caused by the worm [**?**]. [**?**] observed that BGP's keepalive messages may be dropped under heavy congestion and this causes a session to abort temporarily. In all of these cases, neighboring BGP routers must exchange their *entire* routing tables after each session reset. A default-free BGP table typically consists of over 100,000 routes, therefore a table exchange incurs high CPU and bandwidth overhead. This high overhead is particularly unwarranted in the case of a transient failure since most routes in the table are still valid when the session is re-established and therefore do not need to be retransmitted.

The goal of our effort is to *design a fast and bandwidth efficient mechanism* that can detect any inconsistencies between neighboring routers and resynchronize their routing tables whenever inconsistencies are detected. Our approach, Fast Routing Table Recovery (FRTR), uses Bloom filters [**?**] to efficiently encode routing table data and periodically exchanges the Bloom filter digests between BGP peering routers to detect any potential routing inconsistency. After a session reset, FRTR uses digests to identify which routes have changed and sends only those routes, enabling BGP routing to converge much faster.

The use of small digests allows FRTR to maintain low bandwidth overhead, however small digests can also increase the false-positive rate in the Bloom filter digests. To overcome this limitation, we "salt" the digests with random seeds to achieve strong consistency.

Our results show that, in just one round of digest exchange, FRTR can detect more than 91% of random errors and the overhead can be as low as 1.3% of a full routing table exchange; a slight increase in the digest size can achieve a recovery rate higher than 97%. Furthermore, the use of salted digests allows FRTR to achieve error-free routing tables with a probability close to 100% after only a few rounds of digest exchanges. By comparison, the current BGP would not detect any of these errors and even if the errors could be detected, BGP would require a full table exchange to recover.

The remainder of the paper is organized as follows. Section **??** describes our Fast Routing Table Recovery (FRTR) design. Section **??** gives more details about the protocol and implementation. Section **??** shows an example of how FRTR works. To evaluate our design, we use routing tables collected from the Internet and simulate the impact of

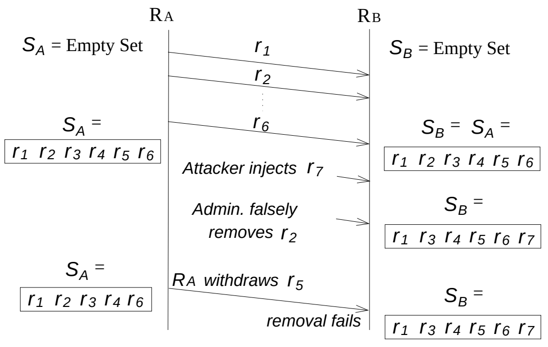

RA
RB
S_A = Empty Set
S_B = Empty Set
r_1
r_2
r_6
S_A =
r_1 r_2 r_3 r_4 r_5 r_6
S_B = S_A =
r_1 r_2 r_3 r_4 r_5 r_6
Attacker injects r_7
Admin. falsely removes r_2
S_B =
r_1 r_3 r_4 r_5 r_6 r_7
S_A =
r_1 r_2 r_3 r_4 r_6
R_A withdraws r_5
removal fails
S_B =
r_1 r_3 r_4 r_5 r_6 r_7

Fig. 1. An Example of Routing Faults

FRTR, as described in Section **??**. Section **??** and **??** present the results. Section **??** describes the related work and Section **??** concludes the paper.

II. FRTR Design

A. Background and Definitions

A *BGP route* r consists of a network address prefix ($Prefix(r)$) and a set of path attributes ($Attr(r)$). BGP path attributes include the AS path used to reach the prefix, the next-hop router, and a variety of other attributes defined as part of the BGP specification.

Neighboring BGP routers exchange routing information and store the information in routing information bases (RIBs). If R_A and R_B are two neighboring routers, the set of routes that R_A advertised to R_B is denoted $RibOut_{A,B}$. The set of routes that R_B learned from R_A is denoted $RibIn_{B,A}$.

Ideally, $RibOut_{A,B} = RibIn_{B,A}$. However, faults or attacks may lead to inconsistencies between them. Examples of such faults and attacks include accidental removal of routes by the administrator, memory corruption, failure to remove a stale route and insertion of an invalid route by an attacker. Figure **??** illustrates how R_B's $RibIn$ can become inconsistent with router R_A's $RibOut$.

Let S denote a set of n routes $\{r_1, r_2, ..., r_n\}$. We define the following types of changes to S that can be caused faults or attacks.

- *Insertion* of r_{n+1} into S: $S' = S \bigcup \{r_{n+1}\}$;
- *Modification* of r_i in S: $S' = S - \{r_i\} \bigcup \{r'_i\}$, where $r'_i = (Prefix(r_i), a')$ and $a' \neq Attr(r_i)$;
- *Removal* of r_i from S: $S' = S - \{r_i\}$.

Since we focus on the communication between two routers, we use $RibOut_A$ and $RibIn_B$ in place of $RibOut_{A,B}$ and $RibIn_{B,A}$ respectively.

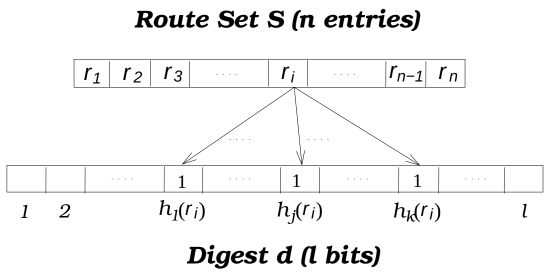

Route Set S (n entries)
r_1 r_2 r_3 r_i r_{n-1} r_n
1 1 1
1 2 $h_1(r_i)$ $h_j(r_i)$ $h_k(r_i)$ l
Digest d (l bits)

Fig. 2. Digestion Computation

B. Design Overview

There are two existing approaches to achieving routing table consistency, each with its own drawbacks. The first approach is to let neighbors periodically send their routes to each other. When the routing table size is big (as in the case of BGP), this brute-force approach has a high cost. The second approach is for a router to compute a checksum for every received route and store both the checksum and the route in its *RibIn*. The router will periodically recompute the checksum of the route and whenever the checksum changes, it requests the neighbor to re-advertise the route. This approach can only detect routing state corrupted by internal errors and offers no protection if a route is removed/inserted along with its checksum or if an obsolete route fails to be removed.

In FRTR, we propose a digest mechanism that unifies the two approaches and at the same time addresses their limitations. A *digest* is a compressed form of a route set. Because each route is only mapped to a few bits in the digest, letting neighboring routers exchange the digest instead of the raw routes can significantly reduce both delay and bandwidth overhead. FRTR takes a soft-state approach and sends routing digests periodically to protect against unexpected insertion, removal, or corruption of the routing state.

In the following sections, we first describe how the digest mechanism works in one round of message exchange. We then show how periodic digest messages with changing "salt" values ensure continued consistency between neighboring routers. Finally, we show how routers can efficiently synchronize their routing tables after a session reset.

C. FRTR Digest Exchange Steps

Step 1: Computing the Sender Digest d_A

The sender, R_A, computes a digest d_A over $RibOut_A$ and sends the digest to neighbor R_B. The digest is a Bloom filter obtained by applying multiple hash functions to each element in a set [**?**].

Suppose we use an l-bit digest (d) and k hash functions ($h_1, h_2, ..., h_k$) to encode a set of n routes (S). Let $d(i)$ denote the i'th bit in d. The digest is initially set to all zero. For each route $r \in S$, we first compute the k hash values $h_1(r), h_2(r), ..., h_k(r)$ and then set the corresponding bits in the digest to be 1, i.e. $d(h_i(r)) = 1$ for $1 \leq i \leq k$. For example, if we use 3 hash functions and the hash values of a route are 65, 39 and 125, then we set the 65'th, 39'th and 125'th bits of the digest to be 1. Note that the hash values may map to bits that have already been set to 1 and those bits will remain to be 1. Figure **??** illustrates how the digest is computed.

Step 2: Identifying Invalid Routes

The neighboring router, R_B, receives d_A and uses it to determine if its $RibIn_B$ contains any routes not currently used by R_A, i.e. the set difference $(RibIn_B - RibOut_A)$.

To identify whether a given route r in $RibIn_B$ belongs to $RibOut_A$, R_B first computes the k hash values of r and then checks whether the corresponding bits in the digest d_A are set to 1. If any of the bits is 0, R_B can conclude that $r \notin RibOut_A$. Otherwise, there is a high probability (but not certainty) that r belongs to $RibOut_A$. In other words, this test does not produce false negatives, but it can produce false positives.

Using this approach, R_B checks every route r in $RibIn_B$ against d_A and places r into one of the following two groups:

1) **Invalid Routes**: $\exists i, 1 \leq i \leq k$, s.t. $d_A(h_i(r)) = 0$. Since Bloom filter does not produce any false negatives, R_B can be certain that $r \notin RibOut_A$ (i.e. $r \in (RibIn_B - RibOut_A)$). These routes may have been inserted or modified.
2) **Probable Valid Routes**: $\forall i, 1 \leq i \leq k$, $d_A(h_i(r)) = 1$. It is probable that $r \in RibOut_A$, but r could also be a false positive. In other words, it is still possible that $r \notin RibOut_A$.

In the above process, R_B also computes its own digest d_B over the group of probable valid routes. More specifically, whenever it identifies a probable valid route, it sets the corresponding bits in the digest.

The false positive rate of a Bloom filter is determined by the ratio between the filter size l and the set size n as well as by the number of hash functions k. It can be computed as follows:

Let p denote the probability that $d(i) = 0$ after the digest is computed.

$$p = (1 - \frac{1}{l})^{k \times n} \approx e^{-k \times \frac{n}{l}} \tag{1}$$

The false positive rate f is the probability P($\forall i$, $1 \leq i \leq k$, $d(h_i(r)) = 1$) where r is not a member of the set in question, i.e.,

$$f = (1-p)^k \approx (1 - e^{-k \times \frac{n}{l}})^k \tag{2}$$

l/n is the *encoding ratio* and is denoted α. It can be proven that f is minimal when $k = ln2 \cdot \frac{l}{n} = ln2 \cdot \alpha$. In other words, for a given encoding ratio, there exists an optimal number of hash functions to minimize the false positive rate. For example, when the encoding ratio is 5, the optimal k is 3.47 (in practice either 3 or 4 is used). Alternatively, one can fix the number of hash functions and adjust the encoding ratio to keep the false positive rate below a target value. For example, if the number of hash function is 3, one can use an encoding ratio of 8 to keep a false positive rate below 3%.

When R_A's Bloom filter has a low false positive rate, R_B can have a high probability of identifying all the invalid routes in $RibIn_B$.

Step 3: Detecting Missing Routes

After removing the invalid routes in Step 2 , R_B now needs to determine whether any routes are missing, i.e. whether $RibOut_A - RibIn_B \neq \emptyset$. One way to test the above hypothesis is to see whether $RibOut_A$ and $RibIn_B$ have the same digest. Note that R_B has already computed its own digest d_B over all the probable valid routes in Step 2.

If $d_A \neq d_B$, we can be certain that $RibOut_A - RibIn_B \neq \emptyset$. However, if $d_A = d_B$, we cannot conclude that all routes from $RibOut_A$ are present in $RibIn_B$ since it is possible that $\exists r$, $r \in RibOut_A$ and $r \notin RibIn_B$, s.t. $d_B(h_i(r)) = 1$ for $1 \leq i \leq k$. The accuracy of this test depends on the false positive rate and the size of $RibOut_A - RibIn_B$; a lower false positive rate and a bigger difference between $RibOut_A$ and $RibIn_B$ both result in higher testing accuracy.

Step 4: Recovering Missing Routes

In this step, we recover the missing routes. Let P_A denote the list of prefixes in $RibOut_A$ and P_B denote the list of prefixes in $RibIn_B$. If $d_B \neq d_A$, R_B sends P_B to R_A. R_A then checks every prefix $p \in P_A$ and classifies p as follows:

1) **Missing Prefix**: If $p \notin P_B$, R_B has no route to this prefix (or had an incorrect route that was removed in Step 2). R_A needs to re-advertise the route to prefix p.

2) **Probable Received Prefix**: If $p \in P_B$, R_B has a route to this prefix and the corresponding path attributes are likely to be correct (since otherwise the route would have likely failed step 2). Nothing needs to be done in this case.

In addition, if $p \in P_B$ and p is not a prefix in P_A, then R_B believes it has learned a route to prefix p even though R_A is not advertising this route. This can occur if the inconsistency was not detected in Step 2 (i.e. it is a false positive). To remove this invalid route, R_A simply sends a BGP withdrawal message to R_B.

D. Periodic Updates

Since one cannot predict how or when a route may be corrupted, error detection and recovery must be done periodically. However, simply resending the entire BGP routing table is infeasible due to its large size. Our approach sends periodic updates that consist of only d_A, the digest of $RibIn_A$ and thus keeps the overhead low. Upon receiving a digest d_A, router R_B begins with Step 2 and rechecks $RibOut_B$.

As with many periodic soft-state refresh schemes, the interval between periodic updates represents an engineering tradeoff. Frequent updates allow routers to quickly catch transient faults, but incur higher bandwidth and processing overhead. On the other hand, infrequent periodic messages require less overhead, but the maximum time before a transient error is detected is increased. Nevertheless, FRTR with a long refresh period is still a qualitative improvement over the current BGP in which unexpected errors stay as permanent errors until the next session reset. Note the trade-off is not necessarily a one-time fixed decision. For example, given a bandwidth budget, [**?**] discusses how to adjust the soft-state rate based on the number of messages to send.

As we mentioned earlier, Bloom filter based digests can lead to false positives, especially when we use small size digest to keep the overhead low. FRTR takes advantage of periodic digest exchanges to overcome this dilemma. In order to catch those false positives, FRTR design uses *"salted"* MD5 hash functions. MD5 [**?**] was chosen since it was designed to be fast and there are several widely available hardware and software implementations. The salt is a randomly generated 32-bit value that is prepended to every route so that the MD5 computation will produce a different signature for the same route in different rounds. The salt can either be negotiated by the two neighboring routers before each round or be carried in every digest. By adding this salt, we use new digests in each periodic exchange which significantly increases the chance that a false positive from one round will not continue to be a false positive in the next round.

E. Routing Table Recovery after Session Reset

Suppose the peering session between R_A and R_B is reset. Let's see how R_B can synchronize with R_A.

First, after the peering session goes down, R_B marks the routes in its $RibIn$ as obsolete. It also starts a timer for the removal of these routes in case the peering session will be down for an extended period of time. Note that the setting of this timer should be negotiated between the two routers so that R_B doesn't prematurely timeout the routes.

When the session comes up, R_A sends R_B its digest and R_B checks if any routes are still valid. If a route can be matched to the digest, its status will be changed from obsolete to valid. At the end of this process, R_B removes any routes still marked as obsolete. Now R_B checks if its digest matches R_A's. If not, R_B sends a request to R_A for missing routes.

This recovery process is very efficient after a transient session failure when only part of the routing table has changed. Instead of blindly sending the entire routing table, R_A sends only the digest and the routes that have indeed changed, reducing both the bandwidth overhead and the CPU processing time. Most importantly, BGP routing convergence will be much faster. The trade-off is that there is a small probability of retaining stale routes due to false positives, but these routes will be removed in the following rounds of checking.

III. Protocol and Implementation Specifics

In the previous section, we presented an algorithm that ensures consistency between $RibOut_{A,B}$ and $RibIn_{B,A}$. In this section, we consider the protocol and implementation issues needed to implement our approach in BGP routers.

A. Route Groups

If we compute a digest over an entire $RibOut$ which has over 120K routes, the digest would exceed the maximum BGP message size and must be sent in a series of fragments which is generally considered undesirable. The digest would be meaningless if any fragment is missing. Moreover, even if the sender could deliver the entire digest reliably, the receiver has to wait till all the individual pieces have arrived before it can start processing the digest.

A better approach is to divide the $RibOut$ into multiple groups by the prefix ranges and then process the routes sequentially in each group, so that the digest for each group of routes can fit into one BGP message. When the sender transmits a digest to the receiver, it also includes in the message the starting and ending prefixes of the corresponding route group. The receiver sorts its routes in the same order. When it receives the digest message, it uses the starting and ending prefix to identify which routes in its $RibIn$ should be matched to the digest. Note that the sender and receiver do not have to agree on how many routes each group has; the decision is entirely left to the sender. The groups do not even have to be the same in size as long as the ir digests can fit into a message.

The above optimization can significantly reduce the bandwidth overhead needed for error recovery. In the straightforward implementation, in order to identify a single missing route, the receiver needs to send the prefix list of all the probable valid rou tes in its $RibIn$ to the sender. Now that each digest only covers a small set of routes, error recovery can be localized to a specific route group and therefore much less information needs to be exchanged.

This optimization requires that the sender and receiver be able to sort their $RibOut$ and $RibIn$. However if the $RibIn/RibOut$ is organized using a Patricia trie structure [**?**], as in the routing software GateD, MRTd and Zebra, there is no need to sort; it is already a binary search tree (the prefixes are used as keys), so an in-order walk of the tree will produce a list of routes sorted by the prefixes. For a $RibIn/RibOut$ containing N routes, the digest can be computed in ti me O(N) as the tree walk visits every route only once. If the BGP implementation cannot sort the routes easily, it can still use FRTR without the optimization.

B. Routing Policies and Path Attributes

In practice, a BGP router may apply its export policy to outgoing routes before advertising them to a peer. The policy may filter out some routes. Therefore, digest computation should be applied to only those routes that match the export policy.

In addition, the peer router may discard some incoming routes according to its import policy. If the sender digest covers all routes sent and the receiver digest includes only routes that matched the import policy, the resulting digests will not match. One possible solution is to use *Cooperative Route Filtering* [**?**] so that the sender only sends those routes that match the receiver's import policy.

Once the sender and receiver agree on which routes will be covered by the digest, an additional problem occurs if the receiver modifies any of the path attributes. For example, the receiver's AS number may be added to the AS path. Other attributes may be modified according to the receiver's import policy, e.g. the receiver may attach a community attribute to a route. If sufficient memory is available, one option is to let the router save a copy of all the pre-policy routes using the "Soft Reconfiguration Inbound" option provided in most BGP implementations.

We propose an approach that uses a new BGP path attribute and eliminates the need to save the pre-policy routes.

In this approach, the sender first uses a hash function to compute a *path attributes hash* that covers only the path attributes associated with the route. This hash is then stored in a new path attribute called $ATTR\_HASH$ and transmitted along with the route announcement. The FRTR digest is then computed using $< salt, prefix, ATTR\_HASH >$. The receiver must store a copy of $ATTR\_HASH$ and must not modify it. However, any other path attributes can be modified or discarded by the receiver. When checking the digest, the receiver computes its digest using its copy of the $< salt, prefix, ATTR\_HASH >$.

Note that to ensure full protection against all types of corruptions, the receiver needs to verify the $ATTR\_HASH$ value when the route is first received. Before modifying or discarding any path attributes, the receiver should compute its own hash over the unaltered path attributes and verify that this hash matches the value in $ATTR\_HASH$. The path attributes can then be modified or discarded. Subsequent digest checks only require the $ATTR\_HASH$ value.

C. Incremental Digest Computation

In the basic design, we recompute all the digests before they are sent. This is necessary because the salt value is changed in every round of checking. To reduce the computation overhead, one may choose to change the salt value less frequently, say every N rounds, and compute the digests incrementally before the salt changes. The tradeoff is that it may take longer time to detect a false positive.

To allow incremental digest computation, one can use a counter for each bit in a digest (see [**?**]). This counter records how many times the corresponding bit has been set to 1. When a new route is added, the bits corresponding to its hash values will be set to 1 and their counters increased by 1. When a route changes, the counters corresponding to the old/new hash values will be decreased/increased by 1. If a counter is 0, the corresponding bit will be set to 0.

The digest does not need to be updated repeatedly if a route is changed multiple times. One can record the old hash values and use a flag to indicate that the route has changed. Then the digest can be updated for all the changed routes just before it is s ent.

D. BGP Messages and Message Order

FRTR defines two new BGP message types: *Digest* and *Prefix*. They both have a common BGP header. A Digest message contains a digest, the hash function and salt value used in the digest computation if they are not pre-negotiated, and the two prefixes to specify the group of routes covered by the digest. A Prefix message contains a list of prefixes whose routes match the peer's digest and the two prefixes to specify the group of routes covered by the digest.

If a sender sends a Digest message before previous route changes have been propagated, the receiver will unnecessarily invoke the recovery process to synchronize its routes. This may happen due to the minimum delay imposed on route changes. BGP updates will be sent only when the MRAI (Minimum Route Advertisement Interval) Timer expires, but the routing table may already reflect all the changes. Therefore, if the digest computation is done on the new routes and the Digest message is sent before the MRAI timer expires, then the receiver will have a mismatching digest. Similarly, a problem will occur if a route changes after its hashes have been computed and the route change is sent before the Digest message is sent. BGP implementations should avoid these problems. More specifically, the Digest message should be sent only after all the existing changes to the corresponding route group have been propagated to the peer(s). Moreover, new changes to the route group should not be allowed while the digest is being computed and those route changes should be sent only after the Digest message is sent.

IV. An Example

From the RIPE RCC00 monitoring point [**?**], we obtained the routing table of a router R_A located in AS2914 dated Jan. 20, 2003, which contains 101,404 routes. We now use it to illustrate how FRTR works assuming that the optimization described in Section **??** is implemented.

Suppose we use a digest size of 1024 bytes and an encoding ratio of 5. These parameters allow each digest to encode $1024 * 8/5 = 1638$ routes and the router R_A can encode its entire table in 62 digests. In each round of consistency checking, R_A sends 62 Digest messages to its peer(s). The generation and transmission of these messages may be paced to avoid congestion.

The first Digest message from R_A covers the routes from 4.0.0.0/8 to 24.240.125.0/24. When R_A's peer at AS7018 (R_B) receives this message, it first locates the starting prefix 4.0.0.0/8 in the corresponding $RibIn$. Then it computes the hash values of every route between 4.0.0.0/8 and 24.240.125.0/24. If any of the routes does not match the digest, that route will be removed. For example, if an attacker injected a forged route for the prefix 4.0.0.0/8, this route most likely will not match the digest and therefore will be removed.

To detect missing routes, the peer R_B computes its own digest in the above process. If the result does not match R_A's, R_B will send a Prefix message to request for missing routes. The Prefix message will contain all the prefixes whose routes match R_A's digest. For example, if the route to prefix 6.1.0.0/16 was mistakenly removed by an administrator, the Prefix message will not contain 6.1.0.0/16. R_A will notice this prefix is missing by comparing the received Prefix message with the list of prefixes in its $RibOut$.

The last Digest message from R_A will carry a flag that indicates the end of all the Digest messages. After processing this message, R_B will remove any routes that have not been matched to any of the received digests.

The operations after a session reset are similar to the above.

V. Evaluation Data and Methodology

A. Data Source

We obtained BGP routing data from the RRC00 monitoring point maintained by RIPE NCC [**?**]. This monitoring point receives BGP routing updates from eleven routers in both large global ISPs and regional ISPs (Table **??** shows their AS numbers and locations), and it periodically archives its routing table.

Location	ASes that RRC00's peers belong to
US	AS7018 (AT&T), AS2914 (Verio), AS3549 (Glocal Crossing)
Netherlands	AS3333 (RIPE NCC), AS1103 (SURFnet)
Switzerland	AS513 (CERN), AS9177 (Nextra)
Britain	AS3549 (Global Crossing)
Germany	AS13129 (Global Access)
Japan	AS4777 (NSPIXP2)
Australia	AS4608 (APNIC)

TABLE I
RRC00's Peering ASes

To obtain the routing table of a monitored router, we simply group the routes in RRC00's routing table according to from which router they were received (i.e. the advertiser's IP address). Our study uses RRC00's routing table archived at 16:00 GMT on Jan. 20, 2003. The number of routes in the eleven derived routing tables ranges from 101,404 to 119,750.

B. Methodology

We simulate two peering routers R_A and R_B. R_A emulates one of the monitored routers of RRC00 by adopting that router's routing table and advertises all the routes to R_B.

In the first part of our evaluation (Section **??**), we assume the BGP session between R_A and R_B fails with a given rate, and compare the bandwidth overhead of FRTR with that of the current BGP. To estimate the latter, we need to estimate the size of a full BGP table exchange. We use a script that scans the BGP table, assembles BGP updates according to the BGP specification and calculates the total size of the updates. The script emulates a BGP

l	α	m	h	k
2^{13} bits	5, 8	1638, 1024	MD5	3

TABLE II
PARAMETER SETTING

implementation that puts consecutive BGP routes into the same update if they share the same path attributes (i.e. only one set of path attributes is sent).

In the second part of our evaluation (Section **??**), we introduce random errors into R_B's $RibIn$ and measure the error recovery power and bandwidth overhead of FRTR. We assign a probability of error P_e and an error type to the routing table, i.e. there is a probability of P_e for generating an error of the given type for each route in the routing table.

Four types of errors are used in our experiments: *removal, insertion, modification and mixed errors*. An error generated for a route r has the following effects on r depending on the error type:

- **Removal**: remove r from the routing table;
- **Insertion**: insert a more specific route of r into the routing table. For example, if r's prefix is 129.250.0.0/16, a route to the prefix 129.250.0.0/17 will be inserted;
- **Modification**: modify r's path attributes;
- **Mixed Errors**: first randomly choose one of the above three types of errors with equal probability, then introduce the chosen error to the routing table.

For simplicity, the path attributes of the inserted or modified route have the same length as those of r and their values are set to 0. Whether the path attributes contain some meaningful values or 0 should not affect FRTR's ability to detect the errors because the hash functions we use have very good dispersion properties and hashing is performed on the *entire route* which contains non-zero bits.

C. Parameter Setting

Table **??** summarizes the parameters used in our experiments. We choose the digest size (l) to be 1,024 bytes or 2^{13} bits so that each Digest message is well within the size limit of BGP messages – 4096 bytes. We use two encoding ratios (α): 5 and 8. Therefore , a digest can encode 1638 routes ($\alpha = 5$) or 1024 routes ($\alpha = 8$). These encoding ratios are not meant to be the optimal values, but are used to illustrate the trade-off between the various performance metrics.

To produce a digest for a group of routes, we first calculate the MD5 signature of each route and then take three 13-bit values (i.e. $k = 3$) from the 128-bit MD5 signature as the hash values.

D. Performance Metrics

We use two performance metrics:

1) *error recovery ratio*: the percentage of errors corrected using FRTR;
2) *bandwidth overhead*: any bandwidth not directly used to correct an error.

In a full BGP table exchange, any BGP routing update that does not correct an error is considered bandwidth overhead and there is no other source of overhead. FRTR uses three types of messages: *Digest*, *Prefix* and *BGP Updates*. The Digest messages and Prefix messages are overhead as they do not directly correct errors.

VI. BGP Session with Transient Failures

In this section, we estimate the long-term bandwidth overhead of FRTR given a session failure rate of λ and compare it with that of the current BGP. Note that routing table inconsistencies caused by other types of faults will not be considered in this section (see the next section for those results).

In FRTR, the bandwidth overhead is the total size of the Digest and Prefix messages. Let's denote them B_d and B_p. We ignore B_p in the following analysis since it is usually much smaller than B_d in the case of transient session failures. If the Digest messages are periodically sent at a rate β, then the bandwidth overhead of running FRTR is $B_d \cdot (\lambda + \beta)$.

In the current BGP, R_A needs to send its routing table to R_B after their session fails. Let's denote the size of this table exchange B_t. Suppose the fraction of routes that actually need to be updated is q, then the overhead is $B_t \cdot (1 - q) \cdot \lambda$.

We are now interested in the ratio $\frac{B_d \cdot (\lambda+\beta)}{B_t \cdot (1-q) \cdot \lambda}$. Using the method described in Section **??**, we estimate that advertising AS2914's entire BGP table would consume $4,980,127$ bytes of bandwidth. The total size of its Digest messages is $65,151$ bytes when the encoding ratio is 5. Therefore, $B_d/B_t = 0.013$ for AS2914. q is usually close to 0 during a transient failure, so the ratio becomes $0.013 \cdot (1 + \beta/\lambda)$.

We can make two observations from the above result:

1) When β is 0, FRTR consumes only 1.3% of the bandwidth overhead of the current BGP in AS2914's case. In other words, FRTR cuts down the overhead of routing table synchronization after session resets by a factor of 77.
2) In the long-term, the overhead of FRTR depends on both the session failure rate and the frequency of periodic Digest messages. Suppose the session fails once a day, FRTR can achieve a lower overhead if the digests are sent once every 19 minutes or less frequently. Note that a session failure rate of once a day is not uncommon.

Some links in operational networks have a failure rate much higher than that (see [?]). Moreover, regular maintenance and policy changes can also lead to session resets.

VII. BGP Table Corruption

In this section, we show the performance of FRTR in recovering corrupted routing tables when a variety of errors are introduced with a wide range of error probabilities. The performance results for one round of recovery are presented in Section **??** and **??**, and the results for multiple rounds are presented in Section **??**.

Since we obtained similar results for all the eleven routing tables, we present only the results for AS2914's routing table here for brevity.

A. Error Recovery Ratio

In Figures **??**–**??**, we show the percentage of errors corrected using FRTR. The X-axis is the probability of error (P_e) in log scale. We have chosen 9 different P_e's in the range of [0.0001, 0.9]. For each P_e, we perform 30 simulation runs, each with a different random seed, to obtain the 95% confidence interval of the mean error recovery ratio. The two curves in each figure correspond to the error recovery ratio for the encoding ratio of 5 and 8, respectively.

These four figures show that, regardless of the error type and the error probability, the higher encoding ratio results in a much higher error recovery ratio. One may also notice that the recovery ratios for removal errors and mixed errors are higher than those of the other two types of errors. We explain the figures in more detail below.

1) Removal Errors: Figure **??** shows that the recovery ratio for removal errors increases from around 92.4% ($\alpha = 5$) and 96.9% ($\alpha = 8$) when P_e is 0.0001 to 100% when P_e is 0.003, and stays at 100% for higher error probabilities. This is because, with only removal errors, all the errors are detected through the "missing routes test", i.e. Step 3 of the digest mechanism (see Section **??**). As P_e increases, more routes are removed from R_B's routing table. This larger difference leads to a higher accuracy in the test.

2) Insertion Errors: The insertion errors show very different characteristics: the error recovery ratio stays around 91% ($\alpha = 5$) and 97% ($\alpha = 8$) regardless of the error probability (see Figure **??**). This is because we evaluate a different step of the digest mechanism here. In this experiment, there are no missing routes so the "missing routes test" is irrelevant. Instead, the inserted routes are detected by checking their hash values against the digest from R_A (i.e. Step 2 of the digest mechanism). The lower the false positive rate with regard to R_A's digest, the higher the percentage of inserted routes detected using this type of checking.

We can compute the false positive rate using Equation **??**. When α is 5, $f = (1 - e^{-k \times \frac{n}{l}})^k = (1 - e^{-k/\alpha})^k = (1 - e^{-3/5})^3 = 0.0918$. The error recovery ratio should be equal to $1 - f \approx 91\%$. When α is 8, the false positive

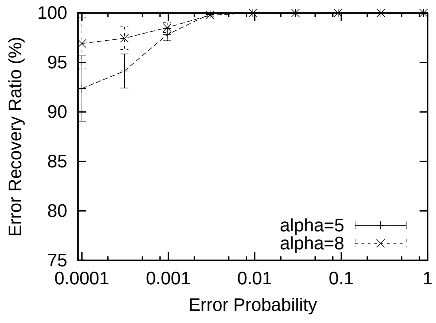


Fig. 3. Recovery Ratio for Removal Errors

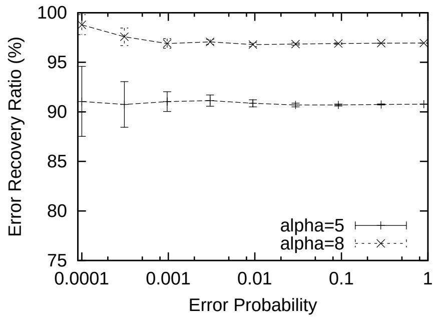


Fig. 4. Recovery Ratio for Insertion Errors

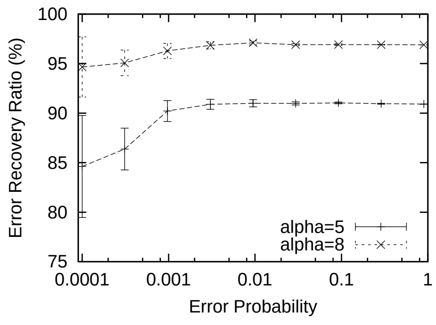


Fig. 5. Recovery Ratio for Modification Errors

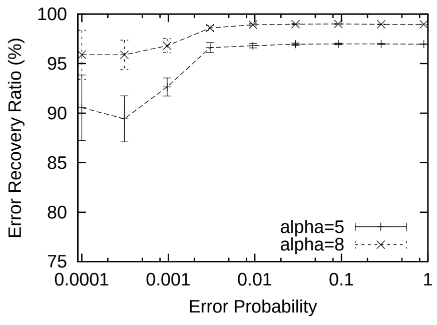


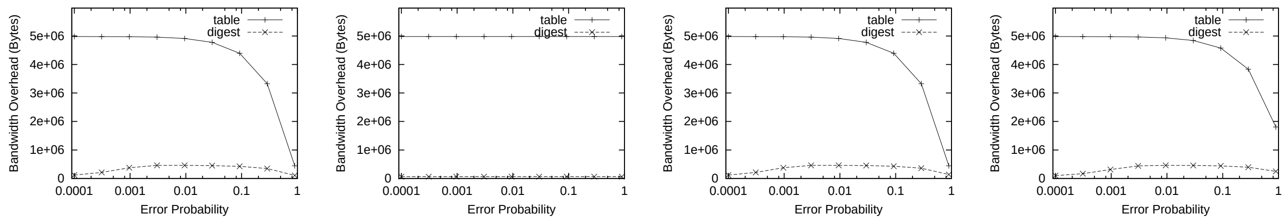
Fig. 6. Recovery Ratio for Mixed Errors

rate f is 0.03 and the error recovery ratio should be roughly 97%. Both numbers match our experimental results. Furthermore, since the false positive rate depends not on the error probability, but on the parameters used in the digest computation (i.e. α and k), the error recovery ratio does not change with the error probability.

3) Modification Errors: Similar to those curves in Figure **??**, the two curves in Figure **??** have an increase at the beginning, and similar to those curves in Figure **??**, they stay around a particular value afterwards (91% for $\alpha = 5$ and 97% for $\alpha = 8$). This is because both Step 2 and 3 are tested in this experiment and the error recovery ratio is affected by the failure rate of both steps.

The initial increase in the two curves is due to the decreasing failure rate of Step 3 as P_e increases (see Section **??**). When P_e is higher than 0.003, the failure rate of Step 3 becomes negligible and the failure rate of Step 2 becomes the major factor in determining the error recovery ratio. As we have shown in Section **??**, the failure rate of Step 2 is determined by the false positive rate in the digest computation. This explains why the two curves stay around 91% and 97% respectively for higher error probabilities.

4) Mixed Errors: Figure **??** shows that when α is 5, the curve increases from around 90% to 97% and stays there, and when α is 8, the curve increases from around 96% to 99% and stays there. We can expect that, if we have a different combination of errors, the curves may move up or down depending on which type of errors is dominant. This is because the result is roughly a combination of the error recovery ratios of the different types of



(a) Removal (b) Insertion (c) Modification (d) Mixed Errors

Fig. 7. Bandwidth Overhead ($\alpha = 5$)

errors.

5) Summary: The error recovery ratio depends on the specific type or combination of errors, as well as the parameters in the digest computation. With a low encoding ratio of five and only three hash functions, we can correct at least 91% of the errors most of the time and achieve a 100% error recovery ratio for removal errors when the error probability is higher than 0.003. Furthermore, increasing the encoding ratio to 8 can significantly increase the error recovery ratio to be around or higher than 97% for most error types and error probabilities.

B. Bandwidth Overhead

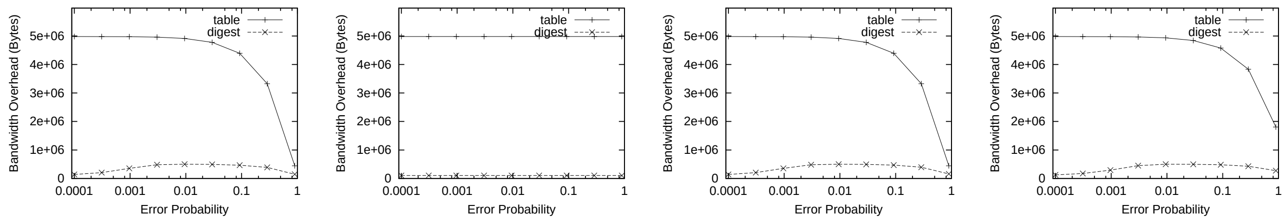
In Figure **??**, we compare the bandwidth overhead of FRTR when α is 5 with that of a full BGP table exchange.

First, we can observe that, in Figure **??**(a), (c) and (d), the curves for a full table exchange show a gradual decrease. This is because the bandwidth overhead is caused by those BGP updates that do not repair erroneous routes and, as the error probability increases, the number of such BGP updates decreases.

Secondly, in Figure **??**(a), (c) and (d), the curves for FRTR first increase with the error probability and then decrease. Since B_d is constant given an encoding ratio and digest size, the non-linearity is caused by B_p (i.e. the Prefix messages). More specifically, as the error probability increases, there are more route groups that contain errors. Each such group produces a Prefix message, so the number of Prefix messages will increase. However, since the number of valid routes in each group may actually decrease, the size of each Prefix message may become smaller. The non-linearity is therefore a result of these two forces.

Finally and most importantly, Figure **??** shows that FRTR with an encoding ratio of 5 has a much lower overhead than a full BGP table exchange for most error probabilities. We explain the difference between the two in more detail below.

1) *Insertion errors* incur the lowest bandwidth overhead (Figure **??**(b) shows the widest gap between the curves). This is because only Digest messages were sent in FRTR to correct the insertion errors, i.e. no Prefix messages



(a) Removal (b) Insertion (c) Modification (d) Mixed Errors

Fig. 8. Bandwidth Overhead ($\alpha = 8$)

were triggered (see Section **??**). The Digest messages consume a constant 65,151 bytes of bandwidth which is only 1.3% of the bandwidth overhead required by the table exchange (4,980,127 bytes). Note that, in the table exchange, R_B corrects all the insertion errors by clearing its $RibIn$ before receiving the full routing table from its neighbor, so none of the BGP updates from R_A during the table exchange serve to correct the insertion errors and they are considered overhead in this case.

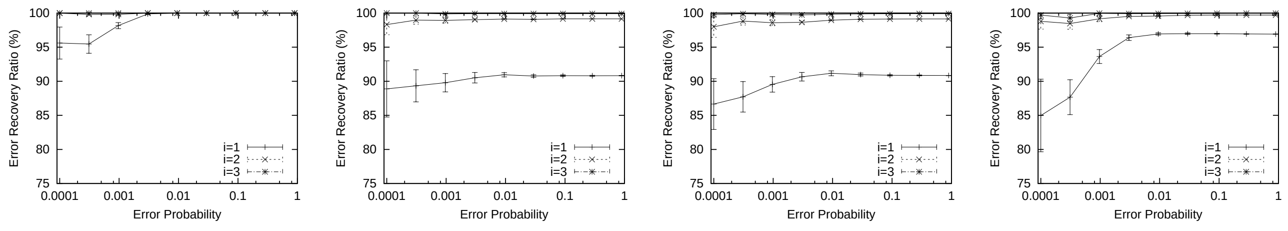
2) For removal errors, the maximum bandwidth overhead of FRTR (460,609 bytes) is reached when the error probability is around 0.009. However, it is still only 9% of the bandwidth overhead of a full table exchange under the same error probability. As the error probability approaches 0.9, the gap between the two curves gets smaller as one would expect, but the full table exchange still has a higher overhead. Modification and mixed errors show similar characteristics as removal errors.

Figure **??** shows that the bandwidth overhead of FRTR with an encoding ratio of 8 is still much lower than that of a full BGP table exchange. The higher encoding ratio increased the bandwidth overhead of FRTR by less than 42K bytes for all error types and error probabilities.

C. Multiple Rounds of Recovery

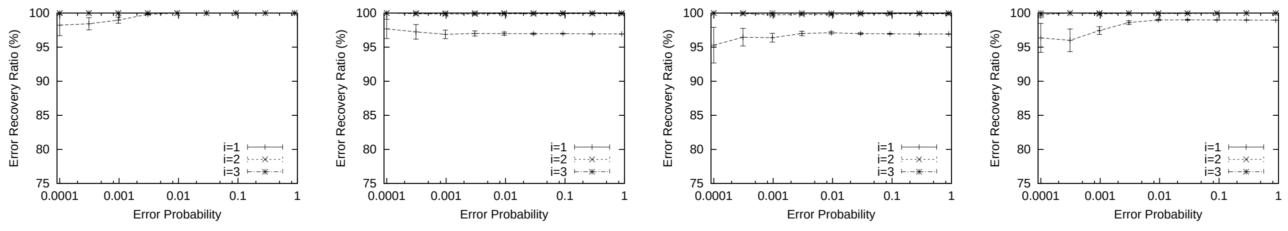
In the previous experiments, we have demonstrated that FRTR can achieve a high error recovery ratio with a low bandwidth overhead after one round of error detection and recovery. However, it is still necessary to let neighboring routers periodically exchange the Digest messages to correct any new errors that have crept in since the last digest exchange, as well as to correct any errors that were left undetected previously, such as those due to Bloom filter's false positive errors.

To evaluate the performance of the periodic "salted" digests, we run FRTR for three consecutive rounds with a range of error probabilities and measure the percentage of corrected errors after each round. For easier understanding of the results, we do not introduce new errors in the second and third round.



(a) Removal (b) Insertion (c) Modification (d) Mixed Errors

Fig. 9. Error Recovery Ratio after Multiple Rounds ($\alpha = 5$)



(a) Removal (b) Insertion (c) Modification (d) Mixed Errors

Fig. 10. Error Recovery Ratio after Multiple Rounds ($\alpha = 8$)

In Figure **??** and **??**, we show the error recovery ratio for multiple rounds of recovery when α is 5 and 8 respectively. The curves labeled "$i = 1$", "$i = 2$" and "$i = 3$" correspond to the results after the first, second and third round. Note that we use a salt even in the first round of error recovery, so the curves labeled "$i = 1$" are slightly different from those in Section **??** since the previous experiments do not use a salt, but the general characteristics of the curves are still the same. We can make the following observations from Figure **??** and **??**:

1) When the encoding ratio is 5 (or 8), we can correct more than 99.9% (or 99.99%) of the errors in most cases after three rounds of error detection and recovery. This matches our analytical result, that is, if the hash values generated in one round are independent from those in the next round, the percentage of uncorrected errors after i rounds should be $(1 - g)^i$ where g is the one-round error recovery ratio (g is approximately 91% and 97% for α of 5 and 8 respectively). This result suggests that, with only a few rounds of salted digest exchanges, we can achieve an error-free routing table with an extremely high probability.
2) The advantage of the higher encoding ratio diminishes quickly with multiple rounds of recovery. A practical implication of this result is that one can use a Bloom filter with a low bandwidth overhead to achieve good long-term performance; although it offers a relatively low one-round error recovery ratio, it can be expected to over-perform the more expensive Bloom filters in the long run.

D. Summary

Our experiments with various error types and error probabilities show that, with only five bits allocated for each routing entry and three hash functions, FRTR can achieve an error recovery ratio higher than 91% most of the time while maintaining a low bandwidth overhead. Increasing the encoding ratio to 8 significantly increases the error recovery ratio while having a small effect on the bandwidth overhead. This may be a good trade-off to make in certain network settings.

Moreover, after only a few rounds of error detection and recovery, FRTR can achieve error-free routing tables with a probability close to 100%, even when the one-round error recovery ratio is relatively low due to the use of a small encoding ratio. Assuming the encoding ratio is an adjustable parameter, this enables network administrators to use relatively inexpensive Bloom filter digests to achieve strong route consistency in the long-run, at the cost of a slightly longer delay in detecting errors.

VIII. Related Work

The Bloom filter was proposed by Bloom in 1970 [**?**] as a space-efficient data structure for membership lookup. It has since been used in operating systems and databases, but had not received wide attention in the networking community until recently ([**?**], [**?**]).

Our work is closely related to [**?**] which proposed several data structures including Bloom filters for approximate reconciliation of set differences. In this paper, Byers, et al. studied the problem of finding as many elements in the set $S_B - S_A$ as possible using a single message. There are two main differences between our work and [**?**]. First, since we are not trying to develop the fastest algorithm for approximate reconciliation of set differences, but to propose a practical solution for a problem in network protocols, we need to be concerned about design issues not considered in [**?**]. For example, to prevent the reassembly of Digest messages from becoming the performance bottleneck, we divide the routing table into groups and compute one digest over each group so that each digest can fit into one routing message. Secondly, we use only a flat data structure (i.e. we do not compute digests over the digests). We are aware of the potential benefit of the hierarchical data structures proposed in [**?**], but we prefer the simplicity of a flat data structure over a hierarchical one for robustness reasons – more complex data structures and algorithms often lead to more errors in design, implementation and operations.

[**?**] proposed a scheme to reduce the overhead of refresh messages in RSVP. The idea is to compute a single digest over all the RSVP state in a node using a tree structure and use the digest to refresh the state in the neighboring nodes. If the digest of the neighboring node matches the received digest, all the state in the neighboring node can be refreshed. Otherwise, the two nodes will start a recovery process to identify the mismatching state entries. In theory, the same scheme can also apply to routing table recovery, but it may not be as easy to use as FRTR. For example, if the nodes need to constantly add or remove state entries, they need to modify their own tree structures in a consistent way. In contrast, senders and receivers do not have such a tight coupling in FRTR, since each Digest message contains the starting and ending prefix of the corresponding group. The recovery process in FRTR is also much simpler than the RSVP refresh reduction scheme due to its flat structure. The latter requires two nodes to walk down the tree structure through message exchanges in order to identify a mismatching state entry, while the former can identify the erroneous entries instantly within each group.

IX. Conclusion

In this paper we presented a scalable technique for detecting and correcting routing table inconsistencies between neighboring BGP routers. Rather than attempting to identify and eliminate *all* the potential errors that might cause inconsistencies, which is an impractical goal in the context of a large scale, distributed system such as the Internet, we take the approach that unexpected faults and errors are inevitable, thus inconsistencies *will* occur. Given that the large size of BGP's global routing table makes the typical soft-state solution of periodic updates infeasible, our Fast Routing Table Recovery (FRTR) design encodes routing table state using Bloom filter digests to effectively and efficiently detect and recover from otherwise unnoticeable errors. The periodic exchange of digests, rather than entire tables, adds only a small cost in overhead. Furthermore, FRTR takes advantage of periodic exchanges by salting the digests, so that the false positive errors, which are inherent in the Bloom filter scheme, can be effectively eliminated after multiple digest exchanges. Finally, FRTR significantly reduces the overhead of routing table recovery after a transient session failure. The current BGP design dictates exchanging the entire routing table after a session reset, while FRTR allows routers to send small digests and the exact set of changed routes.

FRTR design, as described in this paper, focuses on ensuring the consistency between the $RibOut$ of router A and the $RibIn$ of router B. However, a BGP router has several routing tables. In particular, router A has several $RibIn$ tables (one for each interface), a $RibLocal$ table, and several $RibOut$ tables. Although it is equally important to ensure these tables are consistent within a router, such internal consistency can be achieved within a single implementation. But we note that ensuring consistency *between* routers involves products from different hardware manufacturers and/or software versions. Thus we have focused this first effort on the neighboring router consistency; we encourage implementors to apply similar techniques for ensuring consistency within a router's internal tables.

Although our approach has been presented in the context of assuring BGP routing consistency, the techniques we developed here can apply equally well to other protocols where a strong consistency among multiple entities must be enforced.

Finally, this work is part of a broader effort to improve the overall resilience of the global Internet and to understand the principles that lead to resilient protocol design. In future work, we plan to further explore trade-offs with respect to soft-state persistent checking approach and design systems based on what must go right, rather than trying to engineer systems against a set of known faults.