# FRTR: A Scalable Mechanism for Global Routing Table Consistency [*]

Lan Wang, UCLA CSD, lanw@cs.ucla.edu
Daniel Massey, USC/ISI, masseyd@isi.edu
Keyur Patel, Cisco Systems, keyupate@cisco.com
Lixia Zhang, UCLA CSD, lixia@cs.ucla.edu

## Abstract

*This paper presents a scalable mechanism, Fast Routing Table Recovery (FRTR), for detecting and correcting route inconsistencies between neighboring BGP routers. The large size of today's global routing table makes the conventional periodic update approach, used by most routing protocols, infeasible. FRTR lets neighboring routers periodically exchange Bloom filter digests of their routing state. The digest exchanges not only enable the detection of potential inconsistencies during normal operations, but also speed up recovery after a BGP session reset. FRTR achieves low bandwidth overhead by using small digests, and it achieves strong consistency by "salting" the digests with random seeds to remove false-positives. Our analysis and simulation results show that, with one round of message exchanges, FRTR can detect and recover over 91% of random errors that the current BGP would have missed with an overhead as low as 1.3% of a full routing table exchange. With salted digests FRTR can detect and recover all the errors with a probability close to 100% after a few rounds of message exchanges.*

## 1. Introduction

Due to the dynamic and error-prone nature of a network environment, robust neighbor to neighbor communication is an essential part of distributed routing protocols. Earlier routing protocol designs, such as RIP (Routing Information Protocol [12]) and OSPF (Open Shortest Path First [14]), achieved this goal by letting routers periodically exchange their latest routes or connectivity information; any information that is not refreshed will be deleted. Unfortunately, due to the large size of today's global routing table, this periodic update approach is deemed infeasible for BGP (Border Gateway Protocol [17]), the de-facto inter-domain routing protocol used in the Internet. Instead, BGP uses an event driven update approach. Once a router $A$ has sent its initial routing table to a neighbor $B$, it sends no further updates until a route change occurs or the peering session with B breaks. Since routing updates can be lost, re-ordered, or corrupted during transmission, neighboring BGP routers establish a TCP connection and then exchange routing updates over this reliable connection.

However operational experience has shown that reliable update delivery via TCP alone is inadequate to ensure routing consistency between neighbors ([2],[4]). For example, on Oct. 25, 1998, an Internet service provider (ISP) accidentally sent out a large number of invalid routes, creating an outage over large regions of the Internet [2]. The faulty AS quickly withdrew the false routes. However, some of the withdrawn routes were still present in certain areas of the Internet on the following day. Although the exact cause of this specific error is unknown, it is known that earlier BGP implementations by a few major vendors had a common bug which could cause a BGP router to forward a route withdrawal message to some, but not necessarily all, of its neighboring routers [6]. Since the current BGP design does not delete any route until it is specifically withdrawn, stale routes persist in the routing table until some external event (such as a peering session breakdown) flushes out the entire routing table.

Routing state can also be modified or erased by hardware failures or human errors. As far back as in the 1970's, a memory corruption of an ARPANET switch caused an east coast router to falsely announce a zero cost route to UCLA [13]. Because ISPs do not normally report all their routing outages or disclose the exact causes, today it is difficult for the research commu-

[*] 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.

nity to gauge how often routing table corruptions occur in the operational Internet. However, several publicized incidents have been discussed on NANOG, the network operators mailing list (e.g. [8]). There are also numerous examples of configuration errors that led to falsely inserted routes [11]. Moreover, routing state can be altered by malicious attacks[16]. For example, when neighboring routers are connected via a shared medium and their exchanges are not protected by cryptographic mechanisms, another node on the same wire can easily inject a false update. The use of a reliable transport protocol does not protect BGP against any of the above *unexpected* faults.

In addition, BGP suffers from transient peering session failures which can be caused by unstable links or traffic congestion. One study shows that in the Sprint network, physical links between routers go down every 30 minutes on average, and in 80% of these cases the failed links come back in less than 10 minutes [1]. Examination of BGP update logs during the Nimda worm attack in Sept. 2001 showed that some BGP monitoring sessions broke down multiple times, possibly due to the congestion caused by the worm traffic [24]. In these cases, BGP routers cleared all the routes received from their neighbor and then re-sent their *entire* routing tables when the sessions were re-established. Since a default-free BGP table typically contains over 100,000 routes, a table exchange incurs high bandwidth cost and may delay routing convergence. Such a high cost in routing state re-establishment is particularly unwarranted in the case of a transient failure because most routes may still be valid when the session is re-established and therefore do not need to be retransmitted.

The goal of this work 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 filter [3] to efficiently encode routing table data. BGP neighbors periodically exchange their Bloom filter digests to detect any *potential* routing inconsistencies. After a session reset, FRTR uses digests to identify which routes have changed and sends only those routes. In addition, to overcome the false positive drawback of Bloom filter, FRTR "salts" the digests with random seeds and periodically changes the seeds to ensure strong consistency between BGP routers.

We have evaluated FRTR design through both analysis and simulation. Our results show that, with one round of digest exchanges, FRTR can detect and recover 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 detection and recovery rate higher than 97%. Furthermore, the use of salted digests allows FRTR to ensure nearly 100% consistency between neighboring routers after only a few rounds of digest exchanges. By comparison, the current BGP would not detect any unexpected errors; even if the errors were detected, BGP would require a full table exchange to recover. Finally, FRTR facilitates incremental deployment because any two *neighboring* routers can start using FRTR when they both implement the scheme.

The remainder of the paper is organized as follows. Section 2 describes the FRTR design. Section 3 provides more details on the protocol and implementation. Section 4 describes how we used routing tables collected from the Internet to evaluate FRTR. Section 5 and 6 present the results. Section 7 describes the related work and Section 8 concludes the paper.

## 2. FRTR Design

### 2.1. 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 address prefix, the next-hop router, and a variety of other information related to the route.

Neighboring BGP routers exchange routes and store them in Routing Information Bases (RIBs). If $R_A$ and $R_B$ are two neighboring routers, the set of routes that $R_A$ sends to $R_B$ is denoted $RibOut_{A,B}$. The set of routes that $R_B$ learned from $R_A$ is denoted $RibIn_{B,A}$. BGP supports *import and export routing policies*; an *export policy* controls which routes to send to each neighbor and an *import policy* decides which received routes to save and use. We make two assumptions: (a) $R_A$ applies its export policy to a route before putting it in $RibOut_{A,B}$; and (b) $R_B$ stores a received route in $RibIn_{B,A}$ before applying its import policy. Section 3.3 presents solutions when the above assumptions do not hold.

Ideally, we have $RibOut_{A,B} = RibIn_{B,A}$. However, faults or attacks may lead to inconsistencies between them. Examples of such faults and attacks include, but are not limited to, memory corruption, failure to remove a stale route, or insertion of an invalid route. Figure 1 illustrates how $RibIn_{B,A}$ may become inconsistent with $RibOut_{A,B}$. Since we focus on the communication between two routers, we will use the simplified terms $RibOut_A$ and $RibIn_B$ in the rest of the paper.

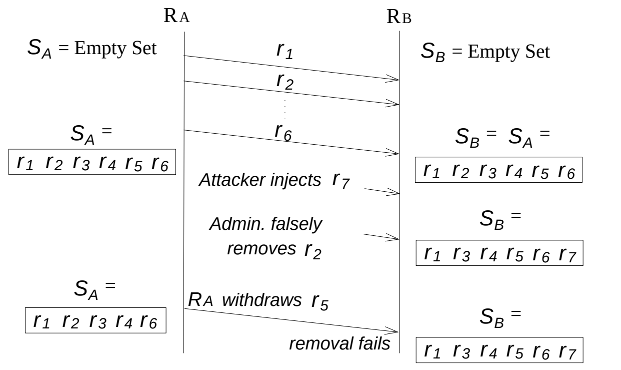


**Figure 1. An Example of Routing Faults**

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 by 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 $S$: $S' = S - \{r_i\}$.

## 2.2. Design Overview

There are two existing approaches to achieving routing table consistency. The first one is to let neighboring routers periodically send their routes to each other. When the routing table size is large, however, this brute-force approach incurs a high cost. The second approach is for a router to compute a checksum for each received route and store both the checksum and the route in its *RibIn*. The router periodically computes a new checksum over the route, and whenever the two checksums do not match, it requests the neighbor to re-advertise the route. This scheme detects unexpected internal changes, such as memory corruption, but offers no protection if a route is accidentally removed along with its checksum, or if an obsolete route fails to be removed.

Our proposal, FRTR, unifies the above two approaches and at the same time addresses their limitations. In FRTR, each router computes a digest over its routes using Bloom filter [3]. Neighboring routers then exchange routing digests periodically to protect against unexpected insertion, removal, or corruption of the routing state. Bloom filter maps each route to only a few bits in the digest, making the periodic exchanges both effective and efficient.

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

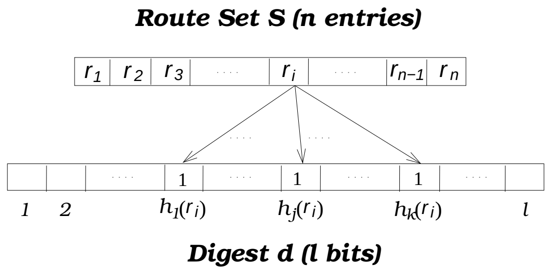


**Figure 2. Digestion Computation**

with changing "salt" values ensure consistency between neighboring routers. Finally, we show how routers can efficiently synchronize their routing tables after a session reset.

## 2.3. 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 $R_B$. Figure 2 illustrates how the digest is computed. Suppose an $l$-bit digest ($d$) and $k$ hash functions ($h_1, h_2, ..., h_k$) are used 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 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 1. Note that the hash values of other routes may map to the bits that have already been set to 1, in such cases those bits simply remain to be 1.

### Step 2: Identifying Invalid Routes

The neighbor, $R_B$, receives $d_A$ and uses it to determine whether its $RibIn_B$ contains any routes not currently used by $R_A$, i.e. the set difference ($RibIn_B - RibOut_A$). More specifically, for a given route $r$ in $RibIn_B$, $R_B$ first computes the $k$ hash values of $r$ and checks whether the corresponding bits in the digest $d_A$ are set to 1. It then 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. $r \in (RibIn_B - RibOut_A)$).

2. **Probably 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.

$R_B$ also computes its own digest $d_B$ in the above process by updating the digest whenever it identifies a probably valid route.

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 $RibIn_B$. The false positive rate of a Bloom filter is determined by the *encoding ratio* ($\frac{l}{n}$) and by the number of hash functions ($k$). It can be computed as follows. First, let $p$ denote the probability that $d(i) = 0$ after the digest is computed.

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

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.,

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

Let's use $\alpha$ to denote the encoding ratio (i.e. $\frac{l}{n}$). It can be proven that $f$ is minimal when $k = ln2 \cdot \alpha$. In other words, for a given $\alpha$, there exists an optimal number of hash functions that minimizes 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%.

### Step 3: Detecting Missing Routes

After removing the invalid routes in Step 2, $R_B$ proceeds 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 probably 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 $RibIn_B$. 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

The goal of this step is to recover the missing routes. Let $P_A$ denote the list of prefixes in $RibOut_A$ and $P_B$ denote the list of prefixes in $RibIn_B$ (excluding the prefixes of invalid routes). 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. **Probably 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 $p \in P_B$ but $p \notin P_A$, we have identified an invalid route in $R_B$. This can occur if the inconsistency was not detected in Step 2 due to a false positive. To remove this invalid route, $R_A$ simply sends a BGP withdrawal message to $R_B$.

## 2.4. Periodic Updates

Since one cannot predict how or when a route may be corrupted, error detection and recovery must be done periodically. The FRTR design lets $R_A$ send periodic updates containing $d_A$ only, the digest of $RibOut_A$, thus keeps the overhead low.

As with all other periodic refresh schemes, the interval between periodic updates represents an engineering tradeoff. Frequent updates allow routers to detect faults quickly, but incur a higher bandwidth and processing overhead. On the other hand, infrequent periodic messages introduce less overhead, but the average time before an 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 permanently until the next session reset. Note the trade-off is not necessarily a one-time fixed decision. For example, given a bandwidth budget, [21] 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 small size digest is used 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 [19] was chosen since its computation is fast and several widely available hardware and software implementations exist. The salt is a randomly generated 32-bit value which is prepended to every route so that the MD5 computation will produce a different signature for the same route when the salt changes. The salt values can be either negotiated by the two neighboring routers beforehand or carried in every digest. Adding the salt enables new digests to be generated in each periodic exchange, which significantly reduces the chance that a false positive from one round would remain as a false positive in the next exchange.

### 2.5. Recovery after a Session Reset

After the peering session between $R_A$ and $R_B$ goes down, $R_B$ marks all the routes in its $RibIn$ as obsolete. It also starts a timer for the removal of these routes in case the peering session remains down for an extended period of time. Note that the setting of this timer should be negotiated between the two routers so that $R_B$ does not prematurely timeout the routes.

When the session comes up, $R_A$ sends $R_B$ its digest and $R_B$ checks whether any routes have become invalid. 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 if $R_B$'s digest still does not match $R_A$'s, it sends a request to $R_A$ for the missing routes.

The recovery process in FRTR is much more efficient than a full routing table exchange because $R_A$ sends only the digests and the routes that have indeed changed during the session down time. More importantly, BGP routing convergence will be much faster. Although there is a small probability of some stale routes are not removed after the first exchange due to false positives, these routes will be removed in the following rounds of checking.

## 3. Design and Implementation Specifics

### 3.1. New BGP Messages

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

### 3.2. Performance Optimization

If we compute a digest over an entire *RibOut* which has over 120K routes, the digest would exceed the BGP message size limit and must be sent in a series of fragments. This is generally considered undesirable because 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.

This optimization can significantly reduce the bandwidth overhead needed for error recovery; since each digest only conveys information of a small set of routes, error recovery can be localized to a specific route group and therefore much less information needs to be exchanged. However, this optimization requires that the sender and receiver be able to sort their *RibOut* and $RibIn$. But explicit sorting is not needed if BGP implementations organize their routing tables using a Patricia trie structure [20], as in the routing software GateD [9], MRTd [15] and Zebra [25], since an in-order walk of the tree will produce a list of routes sorted by the prefixes.

Another optimization is incremental digest computation. In the basic design, we recompute the digests before they are sent 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 longer time to detect a false positive. The details of this optimization are described in [22].

### 3.3. Policy Related Issues

A router may discard some of the routes received from its peer according to its import policy. If it computes a digest over only the saved routes, this digest will not match the peer's. One solution is to use *Cooperative Route Filtering* [7] so that the sender sends only those routes that match the receiver's import policy. Moreover, the router may modify some of the received routes according to its import policy, e.g. attach a community attribute to a route. Such modification will also lead to digest mismatch. We recommend turning on the "Soft Reconfiguration Inbound" option provided in most BGP implementations; this option lets a router save a copy of all the pre-policy routes. In [22], we describe a solution that does not require saving the pre-policy routes.

## 4. Evaluation Data and Methodology

We obtained BGP routing data from the RRC00 monitoring point maintained by RIPE NCC [18]. This monitoring point receives BGP routing updates from eleven routers in both large global ISPs and regional ISPs (Table 1 shows their AS numbers and locations).

| 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 1. 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.

We use a script to emulate two peering routers $R_A$ and $R_B$. $R_A$ adopts one of the routing tables obtained from the RRC00 monitoring point and advertises all the routes to $R_B$. In the first part of our evaluation (Section 5), 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. In the second part of our evaluation (Section 6), we introduce random errors into $R_B$'s *RibIn* and measure the error recovery ratio 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.

### 4.1. Parameter Setting

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 ($\alpha$): 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 MD5 signature as the hash values.

### 4.2. Performance Metrics

We compare the *bandwidth overhead* of FRTR with that of a full table exchange. In FRTR, the Digest and Prefix messages are overhead as they do not directly correct errors. In a full BGP table exchange, any BGP routing update that does not correct an error is considered bandwidth overhead. We also measure the *error recovery ratio* of FRTR. More specifically, we calculate the percentage of errors that are corrected after each round of digest exchange.

## 5. BGP Session with Transient Failures

In this section, we estimate the long-term bandwidth overhead of FRTR given a session failure rate of $\lambda$ and compare it with that of the current BGP. Note that routing table inconsistencies caused by other types of faults will be considered in the next section.

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 $\beta$, 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 $\frac{B_d \cdot (\lambda+\beta)}{B_t \cdot (1-q) \cdot \lambda}$. Let's use AS2914 as an example. We estimate that $B_d/B_t =$

0.013 for AS2914, since advertising its entire BGP table would consume 4,980,127 bytes of bandwidth and the total size of its Digest messages is 65,151 bytes when the encoding ratio is 5. $q$ 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 this result:

1. When $\beta$ is 0, FRTR consumes only 1.3% of the bandwidth overhead of the current BGP in AS2914's case. In other words, if used only after session resets, FRTR cuts down the overhead of routing table synchronization by a factor of 77.
2. When $\beta$ is non-zero, 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 [10]). Moreover, regular maintenance and policy changes can also lead to session resets.

## 6. BGP Table Corruption

In this section, we show the performance of FRTR in recovering corrupted routing tables when a variety of errors are introduced. The performance results for one round of recovery are presented in Section 6.1 and 6.2, and the results for multiple rounds are presented in Section 6.3. Since we obtained similar results for all the eleven routing tables, we present only the results for AS2914's routing table here for brevity.

### 6.1. Error Recovery Ratio

In Figures 3–6, 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 encoding ratio of 5 and 8 respectively.

**6.1.1. Removal Errors** Figure 3 shows the recovery ratio for removal errors. When $P_e$ is 0.0001, the recovery ratio is around 92.4% ($\alpha = 5$) and 96.9% ($\alpha = 8$). Both curves increase to 100% when $P_e$ reaches 0.003, and stay 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

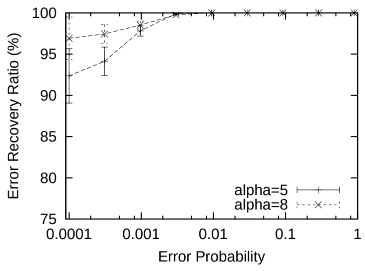


**Figure 3. Recovery Ratio of Removal Errors**

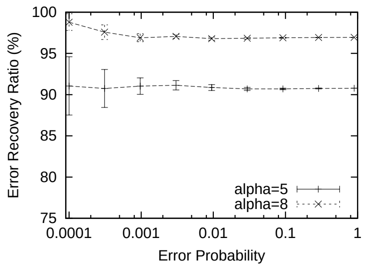


**Figure 4. Recovery Ratio of Insertion Errors**

the digest exchange process (see Section 2.3). 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.

**6.1.2. 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 4). This is because we evaluate a different step of the digest exchange process 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 2. When $\alpha$ 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 $\alpha$ is 8, the false positive 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. $\alpha$ and $k$), the error recovery ratio does not change with the error probability.

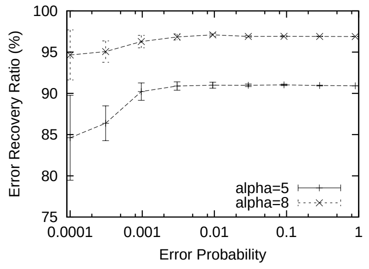


**Figure 5. Recovery Ratio of Modification Errors**

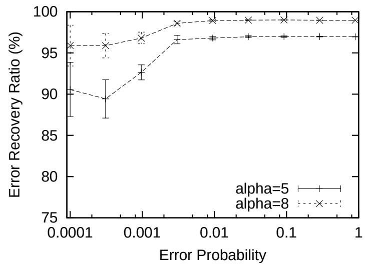


**Figure 6. Recovery Ratio of Mixed Errors**

**6.1.3. Modification Errors** Similar to those curves in Figure 3, the two curves in Figure 5 have an increase at the beginning, and similar to those curves in Figure 4, 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.

**6.1.4. Mixed Errors** Figure 6 shows that when $\alpha$ is 5, the curve increases from around 90% to 97% and stays there, and when $\alpha$ 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 errors.

**6.1.5. 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.

## 6.2. Bandwidth Overhead

Figure 7 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 greater detail below.

First, *insertion errors* incur the lowest bandwidth overhead (Figure 7(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 were triggered (see Section 6.1). 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).

Secondly, 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.

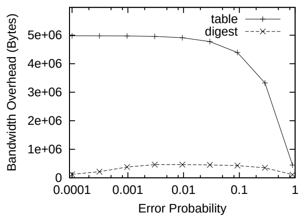
The bandwidth overhead of FRTR with an encoding ratio of 8 is still much lower than that of a full BGP table exchange. The higher encoding ratio increased the bandwidth overhead by less than 42K bytes for all error types and error probabilities. Due to space constraints, we do not show the figure here. Readers may refer to [22] for more details.

## 6.3. Multiple Rounds of Recovery

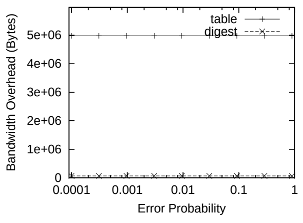
In the previous experiments, we have demonstrated that FRTR can achieve a high error recovery ratio with a low bandwidth overhead after one round of error detection and recovery. However, it is still necessary to let neighboring routers periodically exchange the Digest messages to correct any new errors 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.

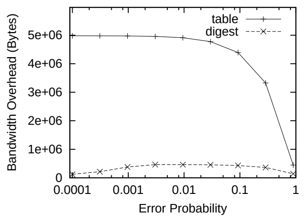
In Figure 8 and 9, we show the error recovery ratio of mixed errors when $\alpha$ is 5 and 8 respectively. The results for other error types are similar (see [22] for the complete results). The curves labeled "$i = 1$", "$i = 2$" and



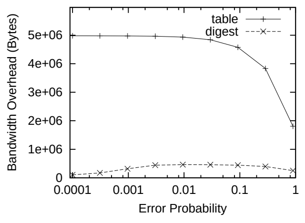
(a) Removal



(b) Insertion



(c) Modification



(d) Mixed Errors

**Figure 7. Bandwidth Overhead ($\alpha = 5$)**

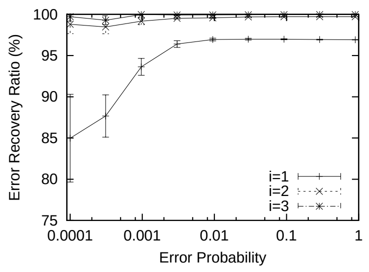


**Figure 8. Recovery Ratio of Mixed Errors after Multiple Rounds ($\alpha = 5$)**

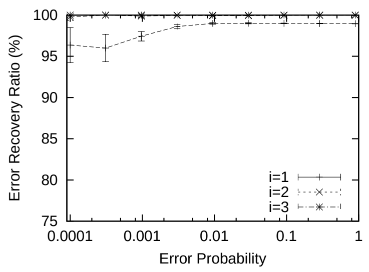


**Figure 9. Recovery Ratio of Mixed Errors after Multiple Rounds ($\alpha = 8$)**

"$i = 3$" correspond to the results after the first, second and third round. The two figures show that, 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 $\alpha$ of 5 and 8 respectively).

We also observe that 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.

## 7. Related Work

The RSVP protocol shares the same challenge as BGP in keeping large amount of state synchronized between neighboring routers. RSVP uses a periodic update approach, which leads to a high overhead as the number of reservations goes up. In [23] we proposed a state compression mechanism to reduce this overhead. The idea is to compute a *single* digest over all the RSVP state in a node using a tree structure and to send this digest, instead of the state information, to refresh the state in neighboring nodes. When the digest indicates an inconsistency, two neighboring nodes need to walk down the tree structure through message exchanges to identify the mismatching state entries. Although the same scheme can be applied to routing table recovery, it requires that neighboring nodes construct and modify their tree structures in a consistent way, a constraint that is difficult to meet across different vendor implementations. In contrast, FRTR only requires that a router be able to put routes in order; each digest message contains the starting and ending prefixes of the routes covered by the digest. The recovery process in FRTR is also much simpler. Due to its flat structure FRTR can identify the erroneous entries instantly within each route group.

Our work is closely related to [5] in which Byers et. al. proposed several data structures including Bloom filter for approximate reconciliation of set differences (we had independently started FRTR design before noticing their work). There are two main differences between FRTR and [5]. First, since FRTR addresses a problem in an existing network protocol, it faces design issues not considered in [5]. For example, to prevent the reassembly of large digest messages, FRTR divides the routing table into groups and computes one digest over each group to make each digest fit into one message. Secondly, although the hierarchical data structures proposed in [5] have certain advantages over the basic Bloom filter, we prefer the simplicity of the latter because more complex data structures and algorithms tend to cause more errors in implementation and operations.

## 8. Conclusion

The large scale of today's Internet presents a fundamental challenge to the design of a resilient global routing protocol. In this paper we present Fast Routing Table Recovery (FRTR), a scalable mechanism for detecting and correcting route inconsistencies between neighboring BGP routers. FRTR encodes routing table state using Bloom filter to efficiently detect and recover otherwise unnoticeable errors. Furthermore, FRTR takes advantage of periodic exchanges by salting the digests to effectively eliminate false positives. Finally, FRTR can significantly reduce the overhead of routing table recovery after BGP session failures.

Although our design has been presented in the context of ensuring routing state consistency *between* BGP neighbors, the same techniques we developed in this work can also be used *within* a router to ensure the consistency between its internal routing table and forwarding table. Furthermore, the basic approach developed in FRTR should be applicable to other protocols where a strong state consistency among multiple entities must be enforced.

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 a resilient protocol design. We plan to further explore the trade-offs between the overhead and the gains of persistent checking, and exploit protocol specific properties to make it more efficient.

## References

[1] S. Bhattacharyya and G. Iannaccone. Tutorial: Availability and survivability in IP networks. In *ICNP 2003*, Nov. 2003.

[2] P. G. Bilse. Re: Is Qwest leaking routes? `http://www.merit.edu/mail.archives/nanog/1998-10/msg00841.html`.

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

[4] V. J. Bono. 7007 explanation and apology. `http://www.merit.edu/mail.archives/nanog/1997-04/msg00444.html`.

[5] J. Byers, J. Considine, and M. Mitzenmacher. Fast approximate reconciliation of set differences. Technical Report 2002-019, Boston University Computer Science Department, 2002.

[6] R. Chandra. Re: Andrew Partan: Re: Mai.net filters. `http://www.merit.edu/mail.archives/nanog/1997-04/msg00480.html`.

[7] E. Chen and Y. Rehkter. Cooperative route filtering capability for BGP-4. *Work in Progress*, Feb. 2003.

[8] S. Donelan. Router crash unplugs 1m Swedish Internet users. `http://www.merit.edu/mail.archives/nanog/2003-06/msg00491.html`.

[9] GateD routing software. `http://www.gated.org`.

[10] G. Iannaccone, C. Chuah, R. Mortier, S. Bhattacharyya, and C. Diot. Analysis of link failures in an IP backbone. In *Proceedings of the ACM SIGCOMM Internet Measurement Workshop 2002*, Nov. 2002.

[11] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP misconfiguration. In *Proceedings of the ACM SIGCOMM*, Aug. 2002.

[12] G. Malkin. RIP version 2. *RFC 2453*, Nov. 1998.

[13] J. M. McQuillan, G. Falk, and I. Richer. A review of the development and performance of the ARPANET routing algorithm. *IEEE Transactions on Communications*, 26(12):1802–1811, 1978.

[14] J. Moy. OSPF version 2. *RFC 2328*, Apr. 1998.

[15] Multi-threaded routing toolkit (MRT). `http://www.mrtd.net`.

[16] S. Murphy. BGP security vulnerabilities analysis. *Work in Progress*, June 2003.

[17] Y. Rekhter and T. Li. A border gateway protocol (BGP-4). *RFC 1771*, Mar. 1995.

[18] RIPE Routing Information Service Project. http://www.ripe.net/ris/.

[19] R. Rivest. The MD5 message-digest algorithm. *RFC 1321*, Apr. 1992.

[20] R. Sedgewick. *Algorithms in C.* Addison-Wesley, Reading, Mass., 1990.

[21] P. Sharma, D. Estrin, S. Floyd, and V. Jacobson. Scalable timers for soft state protocols. In *Proceedings of the IEEE INFOCOM '97*, pages 222–9, Kobe, Japan, Apr. 1997.

[22] L. Wang, D. Massey, K. Patel, and L. Zhang. FRTR: A scalable mechanism to restore routing table consistency. Technical Report 030054, UCLA CSD, Jan. 2004.

[23] L. Wang, A. Terzis, and L. Zhang. A new proposal for RSVP refreshes. In *Proceedings of ICNP '99*, pages 163–72, Toronto, Canada, Nov. 1999.

[24] L. Wang, X. Zhao, D. Pei, R. Bush, D. Massey, A. Mankin, S. Wu, and L. Zhang. Observation and analysis of BGP behavior under stress. In *Proceedings of the ACM SIGCOMM Internet Measurement Workshop 2002*, Nov. 2002.

[25] Zebra routing software. `http://www.zebra.org`.