Persistent Detection and Recovery of State Inconsistencies

Lan Wang *, Daniel Massey, Lixia Zhang

University of Memphis, Computer Science Department, Memphis, TN 38152

Colorado State University, Computer Science Department, Fort Collins, CO 80523

University of California, Los Angeles, Computer Science Department, Los Angeles, CA 90095

Abstract

Soft-state is a well established approach to designing robust network protocols and applications. However it is unclear how to apply soft-state approach to protocols that must maintain a large amount of state information in a scalable way. For example the Border Gateway Protocol (BGP) is used to maintain the global routing tables at core Internet routers, and the table size is typically above 180,000 entries and continues to grow over time. In this paper, we propose a novel approach, Persistent Detection and Recovery (PDR), to enable large-state protocols and applications to maintain state consistency using a soft-state approach. PDR uses *state compression* and *receiver participation* mechanisms to avoid per-state refresh overhead. We evaluate PDR's effectiveness and scalability by applying its mechanisms to maintain the consistency of BGP routing table between routers. Our results show that the proposed PDR mechanisms are effective and efficient in detecting and correcting route insertion, modification, and removal errors. Moreover, they eliminate the need for routers to exchange full routing tables after a session reset, thus enabling routers to recover quickly from transient session failures.

Key words: reliability, fault tolerance, network state management, soft-state, state consistency, refresh overhead

1 Introduction

The soft-state approach has been widely adopted in designing robust Internet protocols and applications. For example most network routing protocols take this approach. In the Routing Information Protocol (RIP) [1], each router send its routing table to its

* Corresponding author.
Email addresses: lanwang@memphis.edu (Lan Wang), massey@cs.colostate.edu (Daniel Massey), lixia@cs.ucla.edu (Lixia Zhang).

neighbor routers every 30 seconds; if any entry in one's routing table is not refreshed for 180 seconds, it is considered obsolete and removed. In the Open Shortest Path First (OSPF) protocol [2], each router broadcasts a link state update message at least every 30 minutes, even when the link state has not changed. These periodic refresh messages not only automatically recover any lost update messages in the past, but also repair any corrupted state at the receiving router, which could be caused by hardware errors or even deliberate attacks. Moreover, the absence of refreshes for a piece of state signals that the state may have become obsolete, and the router removes any state after it has not been refreshed for an extended period of time. Such automatic state expiration not only helps remove obsolete state due to a link failure or a router crash, but also helps *remove false state inserted by attackers automatically*. In short, a soft-state protocol can recover from errors due to known or even unknown causes.

Despite the simplicity and robustness offered by the soft-state approach, not all the protocol designs adopted this approach. One of the primary reasons is the overhead associated with the periodic transmission of refresh messages. When a protocol or an application must maintain a large amount of state, such periodic refreshes become infeasible. For example, in 2003 a typical backbone router on the Internet needed to maintain 100K+ entries in its routing table, and sending the full table would consume more than 5M bytes of bandwidth (see further discussions in Section 5). The global routing table continues to grow and today a typical backbone router maintains over 180K routes [3]. In addition to the bandwidth overhead, there is also a significant overhead associated with assembling and processing the refresh messages periodically. Applications such as large-scale distributed simulation and multi-party Internet gaming also face a similar challenge in maintaining the consistency of a large amount of state. For instance, soldiers engaged in a large-scale battle simulation need to have a consistent view of the combat environment for the simulation to work correctly. Such simulations may involve a huge amount of state that describes the current locations and status of thousands of soldiers, weapons, vehicles, equipments, etc. The soft-state refresh approach is clearly unsuitable in this context.

Up to now protocols and applications that must maintain a large amount of state have been using a *hard-state* approach, which relies on *reliable* message delivery to establish, modify, and remove state between nodes. All state information is sent once only and is kept at the receiving end indefinitely, until and unless explicit notification is received for any further change or removal. For example, the Border Gateway Protocol (BGP) [4] used for inter-domain routing is such a hard-state protocol. BGP uses TCP as a reliable transport protocol to deliver all the updates between adjacent routers. Upon the establishment of a BGP connection, a router sends its whole routing table to the other end; after that it only advertise *incremental changes* to the neighbor, without using any of the periodic state refresh or state timeout mechanisms commonly employed by other routing protocols. One direct consequence of this design is that **stale routes caused by software bugs and other unexpected faults would persist in the routing table**. For example, a well-known software bug from a major router vendor caused BGP routers to delete route withdrawal messages before they were processed [5].

As a result, routers would keep using those stale routes that should have been removed by the withdrawal messages. Hardware failures (e.g. memory corruption) and malicious attacks can also lead to routing table inconsistencies ([6],[7],[8]).

Our work has two main objectives. First, we would like to propose mechanisms that can efficiently maintain a large amount of state *while achieving the robustness of soft-state refreshes*. Previous work on soft-state protocols either does not directly address the issue of large state space [9,10] or otherwise sacrifices the robustness of the protocol in order to handle the increasing amount of state [11–18] (see Section 6 for more discussion of related work). Our proposed solution, Persistent Detection and Recovery (PDR), follows the soft-state paradigm of state management, and uses *state compression* and *receiver participation* to detect and recover state inconsistencies between nodes *without incurring per-state refresh overhead*. PDR is a generalization of the refresh reduction mechanisms that we proposed in [19] and [20]. The key idea behind PDR is *state compression* – each node compresses its entire state space into a digest and then *periodically* exchanges the digest with neighboring nodes to assure state consistency. Compared to other compression techniques, our state compression techniques have the following two novel features: *(1) the digest supports quick identification and recovery of state inconsistencies in a large state space, and (2) the digest can be incrementally computed as state entries are updated.*

Our second objective is to illustrate, by using BGP as an example, *how different PDR mechanisms can be applied to the same protocol and what are the trade-offs between different mechanisms*. We apply two PDR mechanisms to BGP, each using a different state compression technique. The first mechanism uses the *Bloom Filter* technique [21] and the second mechanism uses the *Digest Tree* technique that we proposed in [19]. The performance metrics used in the evaluation include *error recovery ratio, recovery time, bandwidth overhead, computation overhead* and *storage overhead*.

The remainder of this paper is structured as follows. Section 2 introduces background information on BGP and BGP's route inconsistency problems. Section 3 elaborates the design rationale of our PDR approach. In Section 4, we describe two PDR mechanisms as they are applied to BGP. In Section 5, we present evaluation methodology and the results from our performance evaluation. We discuss related work in Section 6, and conclude the paper with our future work in Section 7.

2 BGP Route Inconsistencies

The Internet is composed of a large number of independently managed Autonomous Systems (or ASes). The Border Gateway Protocol (BGP) [4] is used to propagate reachability information among ASes. BGP is a path-vector routing protocol. Its route distribution and convergence properties are quite similar to those of distance vector routing protocols. However, in order to detect routing loops which are inherent in distance-vector routing, BGP attaches a complete path (called an "AS path") to each route. More specifically, a BGP route contains an address prefix representing the destina-

tion network and an AS path to reach the address prefix, together with a set of path attributes associated with the AS path (e.g. the next-hop router's IP address).

To exchange routing information, two BGP routers first establish a BGP session that runs on top of a TCP connection. The routers then exchange their full routing tables through a series of BGP messages. After the initial routing table exchanges, each router sends only incremental updates for new or modified routes. When a router discovers that it can no longer reach a previous reachable address prefix, it sends a message to its neighbor to withdraw the route. A BGP message may advertise a new route (an *Announcement*), change an existing route (an *Implicit Withdrawal*), or withdraw an existing route (a *Withdrawal*). In addition, BGP routers also use periodic keep-alive messages to detect neighbor router crashes and link failures.

2.1 Definition of BGP Route Inconsistency

Neighboring BGP routers exchange routes and store them in the *Routing Information Bases (RIBs)*. Assuming R_A and R_B are two neighboring routers, we denote $RibOut_{A,B}$ as the set of routes that R_A sends to R_B, and $RibIn_{B,A}$ the set of routes that R_B learned from R_A. BGP supports *export* and *import* 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: (1) R_A applies its export policy before putting routes in $RibOut_{A,B}$; and (2) R_B stores all the received routes in $RibIn_{B,A}$ before applying its import policy.

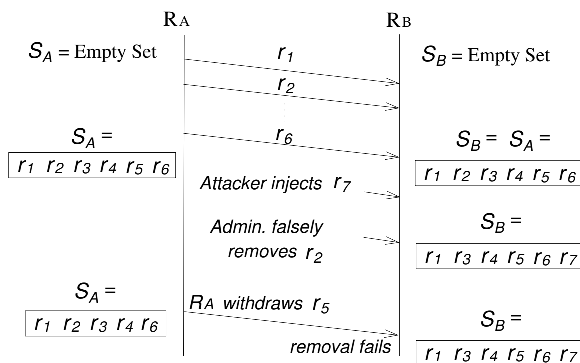

Fig. 1. An Example of Routing Faults.

Ideally, we should 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, and 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 neighboring routers, we use the simplified terms $RibOut_A$ and $RibIn_B$ in the remainder of this paper.

2.2 Causes of BGP Route Inconsistencies

BGP route inconsistencies can be caused by software bugs, hardware failures, and misconfigurations. Routing table corruptions have indeed occurred multiple times in operational networks (e.g. [6,7,22]). Misconfigurations have frequently led to false information being injected into the global routing system. ([23],[24]). Moreover, the current BGP implementation is vulnerable to *attacks*. An attacker may try to inject a false BGP message into the *BGP session*. The false message can change the status of the session, modify or remove existing routes, or even announce false routes, all of which can cause routing inconsistencies ([8,25]).

Furthermore, because each received BGP route is associated with a particular BGP session, a reset of the session leads to removal of all the routes learned over that session, causing the router at each end of the broken session to look for alternative paths for those routes. The Graceful Restart mechanism [26] was introduced to mitigate routing instability during transient session resets. It allows a router to continue to use the routes learned from its neighbor after its BGP session with the neighbor goes down, under the assumption that the physical connectivity still exists between the two BGP neighbors, i.e. the session reset is caused by other types of transient failures. The Graceful Restart does not remove the need for neighbors to exchange their entire routing table once the BGP session is re-established. This table exchange incurs both a high bandwidth cost and a high processing overhead, as well as a delay in repairing routes that may have become stale during the session down time.

Session resets can be caused by congestion [27,28]. In addition, an attacker can reset a BGP session by disrupting the underlying TCP connection [29]. Worse yet, one session reset may be able to trigger a vicious cycle of session resets, as the large volume of BGP messages generated by the table exchange can cause the *keep-alive* messages on other BGP sessions to be dropped. As we will show later in the paper, our PDR design can substantially *speed up* the routing state recovery after a BGP session is up, thus improving the overall network performance either with, or without, the deployment of the Graceful Restart mechanism.

3 Persistent Detection and Recovery (PDR)

Persistent Detection and Recovery (PDR) is designed to ensure state consistency in large-scale systems. It uses soft-state periodic messages to protect against a wide range of faults and attacks. In presenting the main PDR design principles, we first discuss the need for a soft-state approach and the limitations of this approach. We then describe the design of PDR which uses the following two basic approaches to implement soft-state in an efficient way: (a) compressing the large amount of state into a compact form for state refreshes; and (b) utilizing receiver participation in the error detection and recovery process.

3.1 Necessity and Limitations of Soft-State

In our model, a sender and receiver exchange a very large amount of state information over a long period of time. The sender initially announces the state information and may later announce changes, additions, or removals to any part of the state information at any time. The receiver listens for announcements and maintains a copy of the state information learned from the sender. The objective is to ensure that the state information stored at the receiver precisely matches the state information sent by the sender. The problem is challenging due to the potential large-scale of the data and strong fault model in which data may be altered due to faults or attacks. For example, data may be corrupted either in transit, or *after* being received, due to known or unknown causes.

A *hard-state* approach will ensure that state information is correctly recorded at the receiver. After that point, the state is considered to be correctly announced to the receiver and no further updates are sent unless the state changes. There has been much debate over whether soft-state or hard-state is a better approach. Notably, [10] compared several variations of the basic soft-state mechanism with a hard-state mechanism. We agree with one of their findings that the addition of acknowledgment in a soft-state mechanism can improve the state consistency; in fact we proposed adding acknowledgment to RSVP, a soft-state protocol, in [19]. However, we believe that their conclusion of a hard-state approach over-performing a soft-state approach most of the time in terms of state consistency is based on a limited fault model, which does not consider faults and attacks that can remove, insert or modify state at the *receiver* end. Suppose a hard-state protocol has marked some state data as correctly received and no further updates for this data will be sent. At this point, our fault model allows an error at the receiver to corrupt the state. Since there is no further updates for this state from the sender, the invalid state will remain at the receiver indefinitely until the sender changes the state again.

To overcome such faults and attacks, all state must be periodically re-checked using fresh data from the sender. The basic soft-state approach simply resends all state periodically; any corrupted state will be corrected by the next periodic refresh update, and any existing state will be deleted if it is not refreshed after sufficient time elapses. This provides strong protection against faults and attacks. However, when the amount of state information is large, this basic soft-state approach suffers from high refresh overhead due to two design characteristics. First, each piece of state is treated separately and therefore periodic refreshment incurs per-state overhead. Second, the refreshes are open-loop, i.e. the sender sends refresh messages in a blind and brute-force way, without any feedback from the receiver to adjust the transmission for better performance.

The high overhead of soft-state protocols has long been recognized and several solutions proposed. Most of the existing overhead reduction mechanisms ([13–15,30]) try to reduce the refresh overhead by adjusting the refresh timer values. For example, the Scalable Timer mechanisms ([13]) achieves a constant refresh overhead by increasing the refresh interval. Although these timer-based mechanisms can recover from unexpected

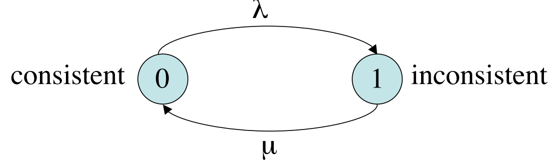

Fig. 2. Markov Chain for Modeling State Consistency between Two Nodes (λ = arrival rate of faults and attacks, μ = arrival rate of refreshes)

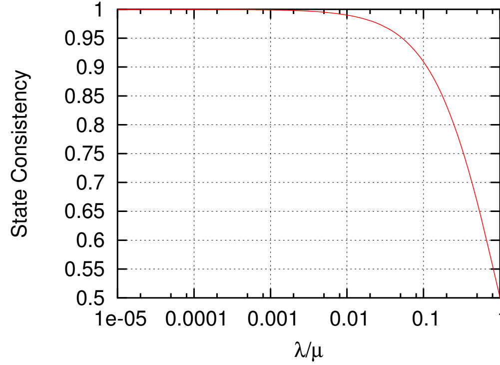

Fig. 3. State Consistency as a Function of λ/μ (λ = arrival rate of faults and attacks, μ = arrival rate of refreshes)

state inconsistencies, the longer refresh interval means longer recovery time in case of unexpected state corruption. As an example, if a BGP table with 3000 routes (about 150K bytes of data) is refreshed every 60 seconds, the resulting bandwidth overhead would be around 20Kbps. *But if the number of routes increases to 100,000 (5M bytes of data), then maintaining the same refresh overhead requires increasing refresh interval to 2000 seconds, more than half an hour.* Assuming the occurrence of faults or attacks is uniformly distributed over time, it will take 1000 seconds on average to correct a state inconsistency, a delay that is intolerable by BGP as well as many other protocols and applications.

Below we show quantitatively that a higher refresh interval leads to lower state consistency. We measure *state consistency* using the percentage of time when two peers have consistent state information (following [9] and [10]). Suppose that faults and attacks strike a state entry e at a rate of λ and refresh messages are sent at a rate of μ (both arrivals follow a Poisson process), we can use a simple Markov chain to model this scenario. The Markov chain has only two modes 0 (consistent) and 1 (inconsistent). It transits from 0 to 1 with a rate of λ (rate of faults and attacks) and from 1 to 0 with a rate of μ (rate of refreshes). The percentage of time when two peers have a consistent value for e is $p_0 = \mu/(\lambda + \mu) = 1/(\lambda/\mu + 1)$. This result means that, *if we increase the refresh interval (μ will decrease and λ/μ will increase), the state consistency p_0 will decrease.*

To further illustrate the effect of increased refresh interval, we plot p_0 in Figure 3. When λ/μ is 0.01, p_0 is around 0.99 (the state is consistent 99% of the time). Now suppose the number of state entries is increased by a factor of 10, but we would like to maintain

the same overhead. In order to keep the refresh overhead constant, we need to increase the refresh interval by 10 times. λ/μ then becomes 0.1 and p_0 will drop to 0.909. If the number of state entries is increased by another factor of 10, the state consistency will further drop to 0.5.

For the above reasons, we do not directly tune the refresh interval in our approach. However, timer adjustment mechanisms can still complement our approach as long as the combination of these different mechanisms can still satisfy the requirements for state consistency and recovery latency.

3.2 The Design of Persistent Detection and Recovery

PDR operates in two phases; a *detection phase* and a *recovery phase*. During normal operation, PDR operates in the detection phase and the periodic digests are used to detect any inconsistencies between the state at the sender and the state at the receiver. A node sends periodic refreshes to its neighbor and each refresh contains a digest of the node's state. If the neighbor's state does not match the digest, PDR enters the recovery stage to identify the out of sync data and restore consistency between the sender and receiver.

3.2.1 Inconsistency Detection Using Soft-State State Compression

A good state compression technique is the key to a PDR mechanism. As discussed above, we require soft-state periodic messages to protect against a wide range of faults and attacks, but traditional soft-state mechanisms suffer from high refresh overhead. To overcome this problem, our PDR first compresses the state to produce a digest and then periodically sends these resulting digests. The main advantage is that the compressed digests require substantially less bandwidth overhead, but an efficient digest alone is not sufficient. This sections identifies the key requirements and benefits. [1]

The most important question our work addresses is how to organize the state in the right structure for compression. The structure must have the following properties:

- The compression technique must dramatically reduce the amount of bandwidth required to send the state. For a large system, sending the uncompressed state with a sufficiently short refresh period requires too much overhead. The primary motivation for introducing the compression is to save bandwidth. But adding compression introduces new computational requirement and new complexity into the soft-state design, thus the bandwidth saving must be substantial enough to both enable the sending of frequent digests and make up for the added cost in complexity.
- The compression technique must be incrementally computable. For a very large-scale system, recomputing the complete digest after any change may place too high a

[1] Note previous techniques for managing refresh timers apply equally well to managing the timers for PDR digests.

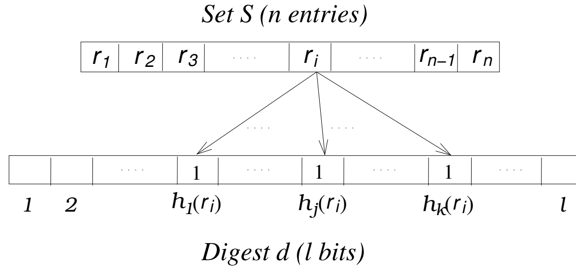

Fig. 4. Computing a Digest using Bloom Filter

computational burden on the sender in generating the digest and on the receiver in comparing the digest.

3.2.2 Inconsistency Recovery Using Receiver Participation

Traditional soft-state protocols operate in an open-loop mode where the sender periodically reports state to the receiver. In PDR, this open-loop mode is again used to send periodic digests to the receiver. However once an inconsistency is detected, PDR uses receiver participation to quickly identify and correct the inconsistent state. By introducing receiver participation, a new advantage is introduced: a PDR digest does not have to carry all the information to *precisely* identify the location of errors. Instead, the digest serves as a launching point for quickly finding the inconsistency using a process where both sender and receiver participate in the identification process.

Based on the basic concepts of periodic soft-state digests and receiver participation in state recovery, we now show how PDR can be used in BGP.

4 Two PDR Mechanisms for Improving BGP Routing Consistency

BGP has to deal with a wide variety of faults and attacks that can remove, modify or insert routes. Robustness against only fail-stop type of failures, such as node crashes and link failures, is not enough to ensure continued functioning of inter-domain routing. Moreover, we need mechanisms that can scale as the BGP table grows. Ten years ago, there were fewer than 20,000 routes in a typical BGP routing table, but in 2006 a default-free BGP table has grown to as many as 180,000 routes [3]. In the remainder of this paper, we present two different PDR mechanisms, **BFilter** and **DTree**, that can satisfy the above requirements.

4.1 Fast Routing Table Recovery with Bloom Filter (BFilter)

In [20], we proposed the FRTR (Fast Routing Table Recovery) mechanism for BGP. It uses Bloom filter [21] for state compression. To distinguish this mechanism from the next one, we refer to it as BFilter.

Figure 4 illustrates how a Bloom filter digest is computed. The basic idea of Bloom filter is to apply multiple hash functions $h_1, h_2, \ldots, h_k$ to every element in a set and mark

the corresponding bits in the digest. Suppose we use an l-bit digest (d) to represent a set of n state entries (S). l/n is called the *Encoding Ratio* and we denote it α. The digest is initially set to all zero. For each $r \in S$, we first compute the k hash values (i.e. $h_1(r), h_2(r), \ldots, h_k(r)$) and then set the corresponding bits in the digest to be 1. More specifically, let $d(i)$ denote the i'th bit in d, then $d(h_i(r)) = 1$ for $1 \leq i \leq k$ after the hashing. Note that some of the hash values of r may map to bits that have already been set to 1, in which case those bits simply remain to be 1.

In the context of BGP, the set S is the BGP table and the elements are BGP routes. We normally choose the encoding ratio α to be a small value, e.g. 5, to lower the storage overhead and bandwidth overhead. For example, with an encoding ratio of 5, a BGP table with 100K routes can be encoded into a digest of 62.5K bytes, which is substantially smaller than the total size of the raw BGP routes (5M bytes in this case). In the remainder of this section, we describe how BFilter works and discuss the trade-offs of various parameter choices.

4.1.1 Operations

Identifying Invalid Routes First, routers periodically send the Bloom filter digests of their outgoing routing table (*RibOut*) to their neighbors. When a router R receives a digest from its neighbor, it examines the routes in its incoming routing table (*RibIn*) associated with the neighbor. If a route r does not match the digest, i.e. $d(h_i(r)) \neq 1$ for any i ($1 \leq i \leq k$), then r is an invalid route and R removes r from its routing table. Otherwise, i.e. r matches the digest, then r is refreshed. More specifically, we associate each BGP route with a cleanup timer, usually set to three times the refresh interval. Refreshing r basically reinitializes the timer. If the cleanup timer expires, this route is removed from the routing table. This mechanism helps removing orphaned or falsely inserted routes.

Detecting Missing Routes After all the invalid routes are removed, R computes a digest over all the remaining valid routes. If this digest still does not match the received digest, R is missing some routes. This step is called the "missing routes test".

Recovering Missing Routes Although R can detect that it is missing some routes, it does not know which routes are missing. Therefore, it requests for the missing routes from its neighbor by sending the prefixes of all the valid routes. The neighbor then replies with all the routes whose prefixes do not appear in R's message. Moreover, the neighbor may detect some prefixes that should not have been in R's list (i.e. "false positives"). In this case, it will withdraw those prefixes.

Recovering from Session Resets When the BGP session goes down, R marks all the routes learned from its neighbor as obsolete so that these routes do not participate in route selection (but they are not removed). When the session is up, it receives a digest instead of a whole BGP table from its neighbor. It then examines the routes in its *RibIn* against the digest. If a route matches the digest, the route's status changes from obsolete to valid. Otherwise, it is removed from the *RibIn*. The router also executes the

"missing routes test" and requests any missing routes from its neighbor. In this way, only new and changed routes are transfered between the routers.

Route Group Optimization If we compute a digest over a *RibOut* that has over 100K 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. In a straight-forward implementation, in order to identify a single missing route, the receiver needs to send the prefix list of all the probably valid routes in its *RibIn* to the sender. By dividing its routing table into multiple route groups, the receiver can associate the missing route with a specific route group and therefore much less information needs to be exchanged.

4.1.2 Performance Analysis

In this mechanism, detecting inconsistencies takes 0.5 round-trip time. If the inconsistencies are caused by injected or orphan routes, no additional recovery time is needed because these routes can be removed once they are detected. However, if there are missing or modified routes, recovering from these inconsistencies will take an additional one round-trip time.

The level of state consistency is directly related to the false positive rate of the Bloom filter. Bloom filter does not produce false negatives, which means that if a route r does not match the digest, the route must be invalid. However, it could produce false positives, i.e. an invalid route may still match the digest. The false positive rate f is a function of the encoding ratio α $(= l/n)$ and 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.

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

The false positive rate f is the probability $\mathrm{P}(\forall i,\ 1 \le i \le k,\ d\big(h_i(r)\big) = 1)$ given that r is not a member of the set in question, i.e.,

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

Equation 2 shows that given the number of hash functions (k), the larger the Bloom filter digest (α), the lower the false positive rate (f). In general, the lower the false positive rate, the higher the probability (q) of detecting and correcting a false route,

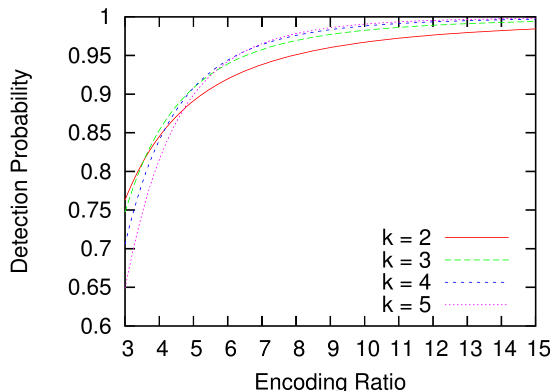

Fig. 5. Probability of Detecting and Correcting a False State using Bloom Filter-based Digest

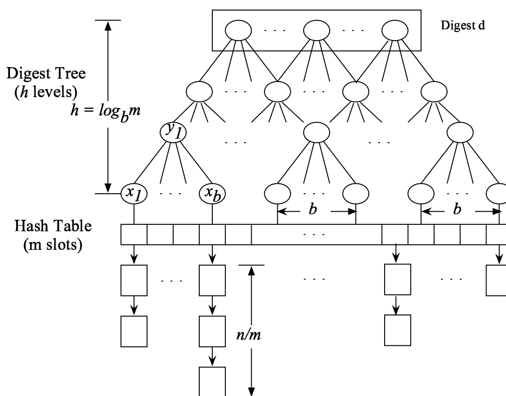

Fig. 6. Computing a Digest using a Digest Tree

i.e. $q = 1-f = 1-(1-e^{-k/\alpha})^k$. We plot the relationship between k, α and q in Figure 5. As you can see, **the detection probability q increases with the encoding ratio α for a given number of hash functions k**. For example, when k is 3 and α is 5, q is 91% (i.e. 91% of the inconsistencies can be detected and corrected). If we increase α to 8, then q is increased to 97%.

4.2 Fast Routing Table Recovery with Digest Tree (DTree)

The DTree mechanism is similar to BFilter, but the digest is computed using a Digest Tree data structure that we proposed in [19]. At the base of the digest tree is a hash table with m slots, each hash slot corresponding to one leaf node of the digest tree (see Figure 6). The tree has a branching factor of b and a height of $h = \log_b m$.

We first hash all the BGP routes into the hash table and compute a checksum over each route. Then we concatenate the checksums of all the routes in each hash slot and compute a checksum for each hash slot. The hash table then produces m level-1 checksums. These checksums are divided into groups of size b and a level-2 checksum is computed over each group, so we get m/b level-2 checksums. This process continues iteratively until there are no more than b checksums in the final set. This final set, i.e.

the root of the digest tree, is called the Digest d of the BGP routes.

4.2.1 Operations

Similar to BFilter, routers periodically send their digests to their neighbors. If a received digest and the local digest match, R refreshes all the corresponding routes in its $RibIn$; otherwise R starts a recovery phase in which the two routers traverse the digest tree to discover and repair the inconsistencies. Below we explain the recovery phase in more detail.

Upon receiving a mismatching digest, R identifies the first mismatching checksum S_1 in the two digests, it then sends an error message containing the lower-level checksums used to compute S_1. The neighbor looks for the first mismatching checksum (S_2) in the error message and sends the children of S_2 in another error message. This procedure is repeated until the checksum (S_{h+1}) causing the problem is found. Then the route corresponding to the checksum will be sent by the neighbor. Note that in this example we assume only one route is the culprit, but in practice multiple routes may be inconsistent and therefore multiple paths in the Digest Tree may be traversed in parallel.

4.2.2 Performance Analysis

It takes 0.5 round-trip time to discover that routing inconsistencies exist. Since the neighbors can compare two levels of the digest tree in each round-trip time, repairing all the inconsistencies takes up to $(h+1)/2 = (\log_b m + 1)/2$ round-trip times. One way to speed up the recovery is to stop at an intermediate level and refresh all those routes that correspond to the problematic checksums at this particular level. In other words, we recover faster by not traversing too far into the tree, but we may increase the bandwidth overhead as some consistent routes may end up being refreshed.

Since the height of the Digest Tree (h) determines the recovery speed, we can also speed up the recovery by increasing the branching factor b thereby decreasing the tree height ($h = log_b m$). However, since the digest size depends on the branching factor, this will increase the bandwidth overhead in both the detection phase and the recovery phase [2].

If we increase the size of each checksum and keep everything else constant, DTree should be able to detect more inconsistencies. More specifically, suppose we insert a false route into R_B's routing table. Let's designate the digest trees computed by the two peering routers $DTree_A$ and $DTree_B$. The Digest Tree technique can detect the false route only if $DTree_A$ and $DTree_B$ have different hashes in all the tree nodes from their root to the particular leaf node. Suppose the hash function has a collision probability of p and the tree has h levels. Assuming that the hashes are independent from each other, the probability that R_B will detect and correct the false route is $q = (1-p)^{h+1}$. Note that the route itself has a hash so there are $h+1$ hash comparisons. We plot q using different

[2] The bandwidth overhead in the recovery phase is linear with respect to $b \cdot (h+1) = b \cdot (log_b m + 1)$, which increases as b goes up when $b \geq 2$.

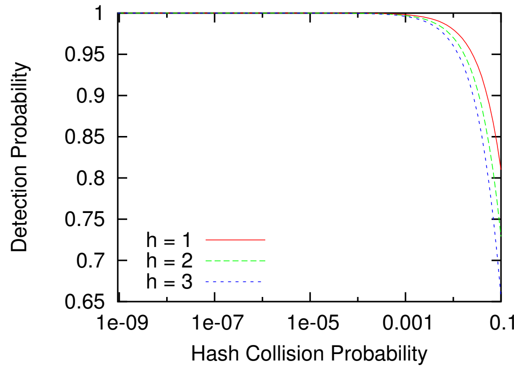

Fig. 7. Probability of Detecting and Correcting a False State using Digest Tree

p and h in Figure 7. As shown in the figure, lower p (collision probability) and lower h (tree height) can both result in a higher detection probability. However, the larger checksum will lead to a higher bandwidth overhead.

4.3 Comparison

The fundamental difference between BFilter and DTree is that the latter uses a hierarchical structure to recursively compute the digest while the former has a flat digest. As a result, DTree can maintain a *constant* bandwidth overhead during the error detection phase regardless of the size of the routing table. BFilter, on the other hand, has a bandwidth overhead that increases linearly with the total number of routes although the overhead is much smaller than sending the raw routing table.

The difference in structure also leads to different behavior during the recovery phase. More specifically, with DTree, one needs to traverse the tree from the root to the leaves in order to recover the individual inconsistencies, while BFilter allows one to recover the inconsistencies within 0.5 to 1.5 round trip times. In addition, the hierarchical DTree digest may be more difficult to implement than the flat BFilter digest.

5 Evaluation

We now evaluate the performance of three instances of the two state compression techniques: (a) a two-level Digest Tree with a branching factor of 110, (b) Bloom Filter with an encoding ratio of 5, and (c) Bloom Filter with an encoding ratio of 8. To simplify our discussion, we denote these three schemes as *DTree*, *BFilter-5* and *BFilter-8* respectively.

5.1 Methodology

We simulate two BGP peers R_A and R_B. R_A advertises its routing table to R_B. We then introduce random errors into R_B's $RibIn$ and measure the error recovery ratio and the overhead of each mechanism. We generate four types of errors: insertion, removal, modification and mixed errors (a combination of the previous three types of errors). An error has the following effects on a route r:

- **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.

We ran the simulation using routing tables obtained from the RIPE RRC00 monitoring point [31]. Since the results from different routing tables are similar, we present only the results for one routing table with 101,404 routes that was used by an operational router in a major US ISP.

To compute a Bloom Filter digest, we first compute a 128-bit MD5 hash over each route and then choose three 13-bit values as the hashes of the route (i.e., $k = 3$). We use a digest size of 1024 bytes and an encoding ratio (α) of 5 or 8. In other words, each digest can encode a group of 1638 routes for $\alpha = 5$ or 1024 routes for $\alpha = 8$ (we adopt the Route Group Optimization mentioned in Section 4.1.1). Because the two encoding ratios produce quite different results in some cases, we present both results.

To compute a digest using the Digest Tree technique, we use a 2-level or a 3-level tree structure with a branching factor between 64 and 110. The digests of the tree nodes are computed using CRC32. Since the results are similar for the two tree levels and the different branching factors, we present only those for the 2-level tree with a branching factor of 110.

Note that the parameters we have chosen are by no means the optimal values. In fact, it is impossible to choose the appropriate values for a particular application without knowing its state consistency requirement, the minimum available bandwidth, the typical error type, the expected error probability and storage availability. However, we do discuss the trade-offs of different parameter values whenever possible.

5.2 Error Recovery Ratio

Figure 8 compares the percentage of errors corrected by the three schemes. 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 to obtain the 95% confidence interval of the mean error recovery ratio. *Note that the current BGP would not detect any of the errors, i.e., its error recovery ratio is 0.*

First, it is clear that DTree has the highest error recovery ratio. In fact, DTree corrected all the errors in all of our experiments, i.e. its error recovery ratio is 100%, regardless of the error type and error probability. This is also true when we use a branching factor lower than 110 and a slightly higher tree structure. In our experiment, we use a two-level digest tree and the collision probability of CRC32 is 2^{-32}, so the detection probability is very close to 1 which is consistent with our analysis in Section 4.2.2.

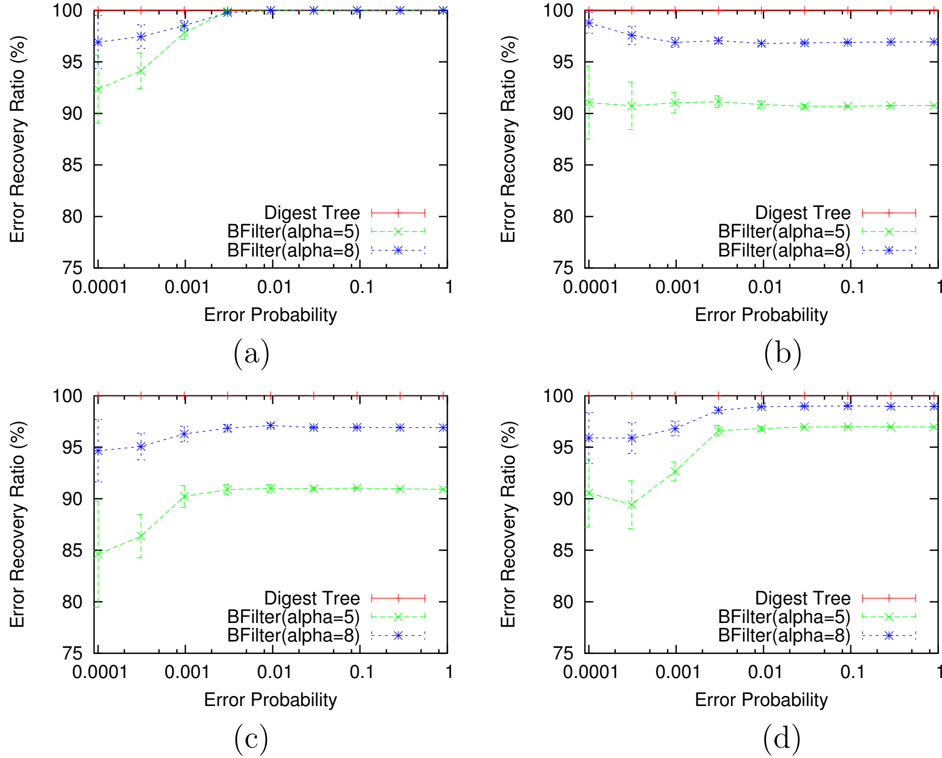

Fig. 8. Error Recovery Ratio: (a) removal errors; (b) insertion errors; (c) modification errors; and (d) mixed errors.

Secondly, Figure 8 shows that BFilter-8 has a higher error recovery ratio than BFilter-5 in most cases. This is because Bloom Filter with a higher encoding ratio has a lower false positive rate, which in general leads to a higher error recovery ratio. Our results again are consistent with the analysis in Section 4.1.2, i.e. the error recovery ratio of BFilter-5 is around 91% while that of BFilter-8 is around 97% except in Figure 8(a) and Figure 8(d) which we explain below.

Thirdly, the curves for the two Bloom-Filter based schemes show different characteristics for different types of errors. There is an increasing trend in the cases of removal errors, modification errors and mixed errors, while the curves for the insertion errors are rather flat. Below we explain the differences.

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". In this test, we compare the digests of R_A and R_B to determine whether R_B is missing any routes. The test can fail when all the missing routes appear as false positives with regard to R_B's digest. 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.

On the other hand, the BFilter5 and BFilter8 curves for insertion errors stay around

91% ($\alpha = 5$) and 97% ($\alpha = 8$) regardless of the error probability. This is because, 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 (let us call it "Bloom filter membership test"). 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.

The curves for the modification errors show an increasing trend at the beginning (similar to removal errors) but they flatten eventually (similar to insertion errors). This is because a modified route is first detected by the "Bloom filter membership test" and removed from the routing table, then we need the "missing routes test" to add the correct route to the routing table. Therefore, the error recovery ratio of modification errors is affected by both the false positive rate of the Bloom filter and the failure rate of the missing routes test.

The figure for mixed errors shows that the Bfilter5 curve increases from around 90% to 97% and stays there, and the Bfilter8 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.

5.3 Bandwidth Overhead

In Figure 9, we show the bandwidth overhead of Full Table Exchange, Digest Tree and Bloom Filter-based Digest. Note that *the bandwidth overhead defined in this study is not the total bandwidth consumption*, but rather the amount of bandwidth consumed by messages that *do not directly* repair errors (since they are unavoidable). For example, a Digest message is considered part of the overhead, while a BGP update used to delete a route after detecting an error is not considered overhead.

The top curve refers to the bandwidth overhead of a full BGP table Exchange. It corresponds to the overhead of the traditional soft-state mechanism. It also represents the overhead of the current BGP when it recovers from a session reset. It is clear that this curve is much higher than the other three curves (except when the error probability is close to 1 in (a) and (c)), which suggests that the PDR mechanisms consume significantly lower bandwidth most of the time. This curve also shows a gradual decrease in (a), (c) and (d). This is because the bandwidth overhead is caused by those BGP updates that do not repair erroneous routes. As the error probability increases, a higher portion of BGP updates in the routing table exchange become "useful" in repairing routes and the bandwidth overhead decreases. The only exception is insertion errors which cannot be repaired by any of the BGP updates directly, so the curve remains high in (b).

In addition, we can also make the following observations. First, we can see that BFilter-5 and BFilter-8 incur similar bandwidth overhead. Second, in Figure 9(a), (c) and (d),

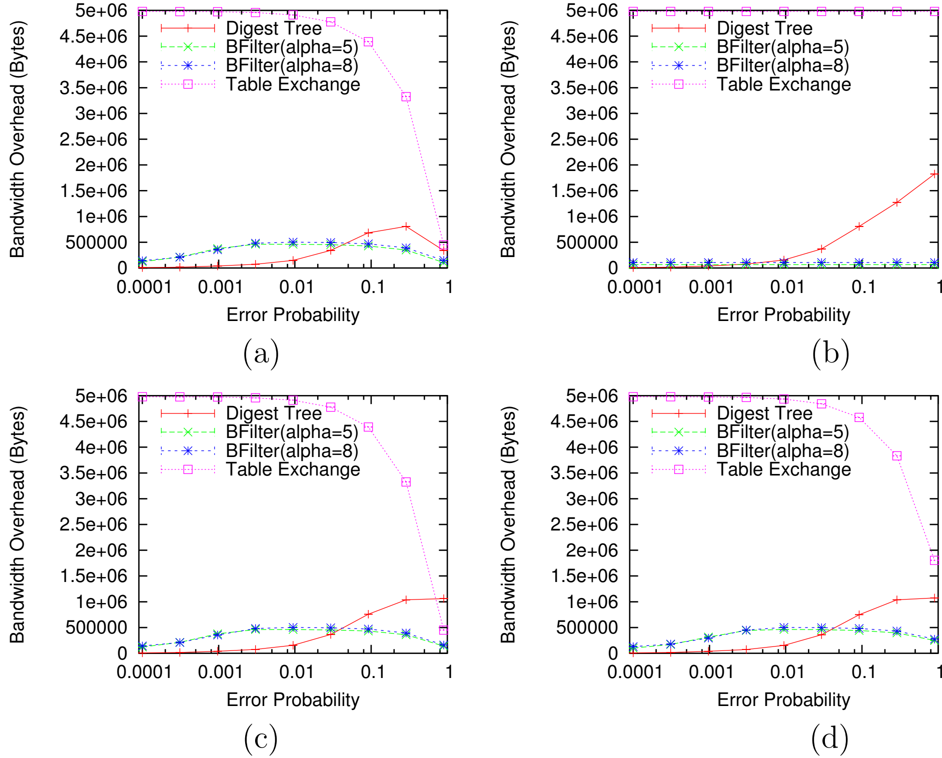

Fig. 9. Bandwidth Overhead: (a) removal errors; (b) insertion errors; (c) modification errors; and (d) mixed errors.

the curve for DTree is lower than the other two curves for low error probabilities, but it eventually rises above the other two curves at an error probability between 0.01 and 0.1. This is because multiple routes can be hashed to the same slot in the hash table. When the error probability increases, it is more likely that at least one route in a hash slot is false. However, in order to identify this one false route, the two peer routers need to compare the hash of every route in that hash slot and each comparison incurs five to eight bytes of overhead (an address prefix + a 32-bit hash). As the error probability approaches 1, the bandwidth overhead will eventually include the hashes in every tree node and the hash of every route.

5.4 Recovery Time

We define recovery time as the period of time from detecting the first error to receiving the message that corrects the last error. The specific recovery time depends on the round-trip time, the message processing time and several implementation details. If we assume that the round-trip time (T) is the dominant factor, then the recovery time of Digest Tree is $(h+1)/2 \cdot T$ and the recovery time of Bloom Filter-based Digest is at most T. Because h is usually a small integer greater than or equal to 2, the Bloom Filter-based Digest technique recovers slightly faster than the Digest Tree technique.

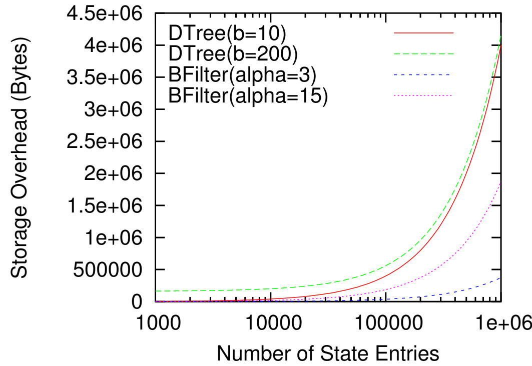

Fig. 10. Comparison of Storage Overhead

5.5 Computation Overhead

Now we estimate the computation overhead of the two techniques. Suppose we want to insert a route to a digest tree. We need to first compute a hash for this route and then update all the hashes from the leaf node (i.e. the hash slot associated with the route) to the tree root. Therefore, we need to compute $h+1$ hashes. On the other hand, Bloom Filter-based Digest requires computing k hashes. Since both h and k are small numbers, the two techniques should have comparable computation overhead if they use similar hash functions.

5.6 Storage Overhead

Finally, we compare the storage overhead of these two techniques. Suppose there are n routes. If each hash is s bytes and each tree node has b children, the storage overhead of Digest Tree is $n \cdot s + b \cdot (b^h - 1)/(b-1) \cdot s$. The storage overhead of Bloom Filter-based Digest is $n \cdot \alpha$. In Figure 10, the two top curves correspond to the storage overhead of Digest Tree with the branching factor 10 and 200 respectively. Both are obtained using a two-level tree structure and a hash size of 4 bytes. The two lower curves correspond to the storage overhead of Bloom Filter-based Digest with the encoding ratio of 5 and 15 respectively. These curves give the typical range of storage requirement that an implementation may have. We can see that the storage overhead of Digest Tree is generally higher than that of Bloom Filter-based Digest.

6 Related Work

Several mechanisms have been proposed to address the soft-state refresh overhead. Most of them fall into the following three categories:

- *Scalable Timer Mechanisms*: these mechanisms adapt the refresh timer of a piece of state *dynamically* to the total amount of protocol state so that the bandwidth used by refresh traffic remains below a threshold. The most notable examples in this category are the dynamic timer adjustment mechanism used by RTCP [11] (SAP uses a similar mechanism [12]) and the “Scalable Timers” mechanism proposed by Sharma et al.

for PIM [13].

- *Staged Timer Mechanisms*: these mechanisms set the initial refresh timer to a small value and then increase the timer by a percentage after every refresh. There is usually an upper bound on the refresh interval. One example of the Staged Timer mechanisms is the variable heartbeat mechanism proposed by Holbrook et al. to achieve scalable and timely distribution of data updates [14]. Pan et al. also proposed, in the context of RSVP, an exponentially increasing refresh timer [15].
- *Hard-state with Keep-alives*: one extreme approach to refresh overhead reduction is to convert a soft-state protocol to a hard-state protocol with *keep-alive* probes between the two nodes. This approach has been proposed for RSVP [16], RIP [17], and OSPF [18].

While the above approaches lessen the refresh overhead problem to various degrees, they tend to result in lower state consistency when the network environment is unpredictable. Their fundamental problem is simple assumptions of the network environment. For example, the "hard-state with keep-alive" approach assumes that as long as the receiver is alive and state updates are reliably delivered, the receiver's state will be consistent with the sender's. This assumption overlooks the possibility that a network fault or attack may corrupt the receiver's state, resulting in a lower level of state consistency.

In [9], Raman and McCanne studied announce-listen protocols [32,33], which they call "soft state-based data communication". These protocols are usually used for *data delivery* in a multicast setting with the assumption that once a piece of **data** is delivered to a receiver the data will remain consistent. As such, the subsequent announcements serve only two purposes: (a) to recover from losses during transmission and (b) to reach *new* multicast receivers. This is why they consider the transmissions of already consistent data items mostly "redundant" and try to assign lower priority to these messages so that other fresh data items can reach the receivers faster. This simple assumption does not hold in our case because of the following reasons: (a) state information is not consumed by the receiver the same way as real-time streaming data – it stays at the receiver side to support certain decision making; (b) a piece of state may be modified or even deleted by faults or attacks at the receiver side so state refreshes are as important as state changes. Therefore the two-level priority scheme that they proposed, i.e. assigning already consistent items a lower priority, cannot be directly applied to the management of state consistency. However, the idea of prioritization still has its application in state management. For an instance, some state entries may be more important than others, e.g. some popular BGP routes receive much more traffic than the others, so it may be desirable to give them a higher priority.

7 Conclusion

In this paper, we address a long-standing problem in protocol design – *how to ensure state consistency in a large-state protocol such as BGP without turning it into a hard-state protocol*? We show that, with our approach "Persistent Detection and Recovery (PDR)", it is possible to follow the soft-state paradigm in an efficient manner. We

have demonstrated PDR's effectiveness in improving BGP routing consistency. Our simulation shows that the two PDR mechanisms, BFilter and DTree, are efficient in correcting route insertion, modification and removal. Moreover, they eliminate the need for routers to exchange full routing tables after a session reset, thus enabling routers to recover quickly from transient session failures. Finally, we have shown that each of the two proposed state compression techniques has its own trade-offs. Therefore, a protocol designer needs to take into consideration all the requirements of his specific protocol to choose the right parameters.

As our next step, we plan to apply PDR to other protocols to improve their state consistency. The proposed PDR mechanisms are designed for BGP and RSVP. In both protocols, there may be a large amount of state shared between neighboring nodes and each node usually has a small number of neighbors. We have not applied PDR to the scenario where each node broadcasts its state to all the other nodes. In addition, we plan to study the scenario where the state on each node changes frequently. The challenge here is to ensure that the recovery process does not interfere with the propagation of new state changes.

References

[1] G. Malkin, RIP version 2, RFC 2453.

[2] J. Moy, OSPF version 2, RFC 2328.

[3] The BGP report, http://bgp.potaroo.net/.

[4] Y. Rekhter, T. Li, S. Hares, A Border Gateway Protocol (BGP-4), RFC 4271.

[5] S. Donelan, Re: Routing loop between Qwest and Sprint?, http://www.merit.edu/mail.archives/nanog/1999-05/msg00157.html (May 1999).

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

[7] S. Donelan, Router crash unplugs 1m Swedish Internet users, http://www.merit.edu/mail.archives/nanog/2003-06/msg00491.html (Jun. 2003).

[8] S. Murphy, BGP security vulnerabilities analysis, RFC 4272.

[9] S. Raman, S. McCanne, A model, analysis, and protocol framework for soft state-based communication, in: Proceedings of the ACM SIGCOMM '99, Cambridge, MA, 1999, pp. 15–25.

[10] P. Ji, Z. Ge, J. Kurose, D. Towsley, A comparison of hard-state and soft-state signaling protocols, in: Proceedings of ACM SIGCOMM '03, Karlsruhe, Germany, 2003.

[11] H. Schulzrinne, S. Casner, R. Frederick, V. Jacobson, RTP: A transport protocol for real-time applications, RFC 3550.

[12] M. Handley, C. Perkins, E. Whelan, Session announcement protocol, RFC 2974.

[13] P. Sharma, D. Estrin, S. Floyd, V. Jacobson, Scalable timers for soft state protocols, in: Proceedings of the IEEE INFOCOM '97, Kobe, Japan, 1997, pp. 222–9.

[14] H. W. Holbrook, S. K. Singhal, D. R. Cheriton, Log-based receiver-reliable multicast for distributed interactive simulation, in: Proceedings of the ACM SIGCOMM '95, 1995, pp. 328–341.

[15] P. Pan, H. Schulzrinne, Staged refresh timers for RSVP, in: Proceedings of the IEEE GLOBECOM '97, Phoenix, AZ, 1997, pp. 3–8.

[16] L. Berger, D. Gan, G. Swallow, P. Pan, F. Tommasi, S. Molendini, RSVP refresh overhead reduction extensions, RFC2961.

[17] G. Meyer, S. Sherry, Triggered extensions to RIP to support demand circuits, RFC 2091.

[18] P. Pillay-Esnault, OSPF refresh and flooding reduction in stable topologies, RFC 4136.

[19] L. Wang, A. Terzis, L. Zhang, A new proposal for RSVP refreshes, in: Proceedings of IEEE ICNP '99, Toronto, Canada, 1999, pp. 163–72.

[20] L. Wang, D. Massey, K. Patel, L. Zhang, FRTR: A scalable mechanism for global routing table consistency, in: Proceedings of the International Conference on Dependable Systems and Networks, 2004, pp. 465–474.

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

[22] University of Alaska, 08-12-04 routing table corruption, `http://technology.uaa.alaska.edu/kb/Incident_Reports/08-12-04-Routing-Table-Corruption.cfm` (Aug. 2004).

[23] X. Zhao, D. Pei, L. Wang, D. Massey, A. Mankin, S. Wu, L. Zhang, An analysis of BGP multiple origin AS (MOAS) conflicts, in: Proceedings of the ACM SIGCOMM Internet Measurement Workshop 2001, 2001, pp. 31–35.

[24] R. Mahajan, D. Wetherall, T. Anderson, Understanding BGP misconfiguration, in: Proceedings of the ACM SIGCOMM, 2002.

[25] B. Christian, T. Tauber, BGP security requirements, Internet Draft (Work in Progress).

[26] S. Sangli, Y. Rekhter, R. Fernando, J. Scudder, E. Chen, Graceful restart mechanism for BGP, Internet Draft (Work in Progress).

[27] A. Shaikh, A. Varma, L. Kalampoukas, R. Dube, Routing stability in congested networks: Experimentation and analysis, in: Proceedings of the ACM SIGCOMM 2000, Stockholm, Sweden, 2000, pp. 163–74.

[28] L. Wang, X. Zhao, D. Pei, R. Bush, D. Massey, A. Mankin, S. Wu, L. Zhang, Observation and analysis of BGP behavior under stress, in: Proceedings of the ACM SIGCOMM Internet Measurement Workshop, 2002, pp. 183–195.

[29] NISCC, Vulnerability issues in TCP, `Vulnerability Issues in TCP` (May 2004).

[30] J. Rosenberg, H. Schulzrinne, Timer reconsideration for enhanced RTP scalability, in: Proceedings of the IEEE INFOCOM '98, 1998.

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

[32] M. Chandy, A. Rifkin, E. Schooler, Using announce-listen with global events to develop distributed control systems, Concurrency: Practice and Experience 10 (11-13) (1998) 1021–7.

[33] S. Floyd, V. Jacobson, S. McCanne, C.-G. Liu, L. Zhang, A reliable multicast framework for light-weight sessions and application level framing, in: Proceedings of the ACM SIGCOMM '95, Boston, MA, 1995.