Identifying BGP Routing Table Transfers [*]

Beichuan Zhang [†]
bzhang@cs.arizona.edu

Vamsi Kambhampati [‡]
vamsi@cs.colostate.edu

Mohit Lad [§]
mohit@cs.ucla.edu

Daniel Massey [‡]
massey@cs.colostate.edu

Lixia Zhang [§]
lixia@cs.ucla.edu

ABSTRACT

BGP routing updates collected by monitoring projects such as RouteViews and RIPE have been a vital source to our understanding of the global routing system. The updates logged by these monitoring projects are generated either by individual route changes, or are part of BGP table transfer. In particular, a session reset between a monitoring station and its BGP peers can result in the peer sending its entire BGP routing table to the monitoring station. In this paper, we present a Minimum Collection Time (MCT) algorithm that accurately identify the start and duration of routing table transfers. Using three months of data from 14 different peers, MCT can identify routing table transfers triggered by BGP session resets with 100% accuracy, and can pinpoint the exact starting time of table transfers in 90% of the cases.

Categories and Subject Descriptors

C.2.2 [**Computer Communication Networks**]: Routing Protocols

General Terms

Measurement, Experimentation, Algorithms

Keywords

BGP, Session Reset, Routing Table Transfer, Collection Time

[*]This work is partially supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. N66001-04-1-8926 and by National Science Foundation (NSF) under Contract No. ANI-0221453. Any opinions, conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the DARPA or NSF.

[†]Computer Science Department, University of Arizona.

[‡]Computer Science Department, Colorado State University.

[§]Computer Science Department, University of California, Los Angeles.

1. INTRODUCTION

The Border Gateway Protocol (BGP) is the *de facto* interdomain routing protocol on the Internet. Numerous projects use BGP update data to analyze Internet routing, diagnose routing problems, and evaluate improvements to BGP. Oregon RouteViews [7] and RIPE RIS [6], the two best known BGP monitoring projects, maintain several monitors that establish BGP peering sessions with routers in many operational networks. These monitors receive and log BGP routing updates from their peers. These BGP update logs, along with data from individual ISPs own collection, are indispensable to researchers and network operators in analyzing global routing dynamics.

BGP updates in the monitor logs can be divided into two general categories, table transfer updates and incremental updates. When a BGP peering session is established, a router advertises to the new peer all the routes that are currently in its routing table and match its export policy. We call these updates *table transfer* updates. After the initial table transfer, the router sends out incremental route changes only[1]. We call these updates *incremental* updates.

It is often important to distinguish between these two types of updates. For example, suppose on a typical day a BGP monitor records a few tens of thousands of BGP updates and on the following day it logs well over a few hundreds of thousands of BGP updates. The implications of such a 10-fold increase in daily update counts heavily depends on whether the updates are due to table transfers or incremental changes. A typical BGP routing table contains over 120,000 routes, thus a single BGP session reset between the monitor and its peer can result in over 120,000 updates. If table transfers are a problem, changes to increase the stability of peering session with the monitors are needed or the update logs need to identify the table transfer updates. On the other hand, if the update jump is the result of a large spike of incremental updates, techniques for improved convergence, route damping, and so forth may be warranted. In general, distinguishing between table transfers and incremental updates can have important implications for a variety of BGP analysis. Unfortunately, some BGP monitors do not log when a session resets. Even if the start of a session reset is logged, it does not indicate how long a table transfer lasts and whether an update is part of a table transfer or an incremental change.

[1]In practice, some BGP routers send duplicate updates that simply re-announce the same route.

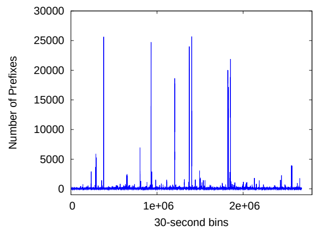

Figure 1: Number of prefixes in every 30 seconds

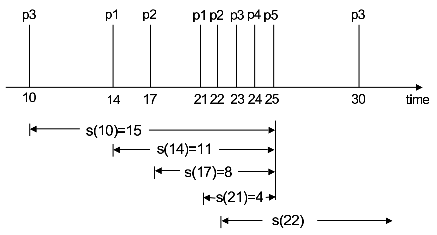

Figure 2: Update Stream and Collection Time

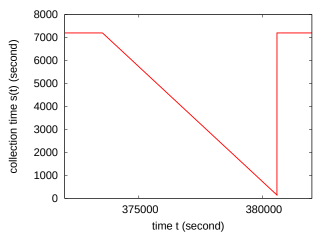

Figure 3: Sample $s(t) \sim t$

In this paper, we present a Minimum Collection Time (MCT) algorithm that accurately detects the start and duration of table transfers from a stream of BGP updates. MCT does not rely on any explicit indication of the occurrence of BGP session resets. Because BGP monitors log all routing updates, those table transfer updates caused by BGP session resets between monitors and their peers are monitoring artifacts rather than real routing dynamics in operational networks, thus it is important to identify and filter them out. Using three months of data from 14 different peers, MCT can identify full routing table transfers triggered by BGP session resets with 100% accuracy, and can pinpoint the exact start of table transfers in 90% of the cases. The MCT algorithm works with common BGP log formats such as those used at RouteViews and RIPE and is available at `http://netsec.cs.colostate.edu/`.

2. THE BASIC APPROACH

Given a stream of BGP updates, our objective is to detect the existence of all table transfers that may have occurred in the update stream. If a transfer does occur, we want to identify its *starting time* (i.e., timestamp of the first update of the table transfer) and its *duration* (i.e., how long it takes to finish the transfer). This section presents the intuition and basic Minimum Collection Time algorithm. The next section addresses a few practical issues in processing real BGP data.

The main characteristic of a table transfer is that, in the update stream, *all* the prefixes in the routing table appear within a short period of time. A simple way to identify table transfers would be to visually look for spikes in the number of prefixes for which updates are generated over time. For instance, Figure 1 shows the number of prefixes contained in update messages in 30-second bins based on one-months update from a router. Although we can visually identify some spikes, it is difficult to pin down their exact starting and ending times. Besides, it is not clear which peaks correspond to table transfers, and over which time interval (i.e., bin width) the number of prefixes should be counted. In order to automatically and accurately identify table transfers, we developed the following approach.

Assume a BGP session was established and a stream of updates is received as shown in Figure 2, and also assume that the entire routing table consists of routes to five prefixes only, $p1$, $p2$, $p3$, $p4$, and $p5$. The updates at time 10, 14, and 17 are *incremental updates* announcing a change in the route to prefix $p3$, $p1$, and $p2$, respectively. A table transfer happens at time 21 and ends at time 25, during which the routes to all five prefixes are announced. The resulting updates are table transfer updates and this table transfer lasts $25 - 21 = 4$ seconds. The update at time 30 is an incremental update.

For an update received at time t, we define its ***collection time***, $s(t)$, as the time it takes *for all prefixes to be announced*. For example, consider the update that arrives for $p3$ at time 10. Starting at time 10, it takes until time 25 for all five (unique) prefixes to be announced, and thus $s(10) = 25 - 10 = 15$. Similarly, $s(14) = 25 - 14 = 11$, $s(17) = 8$, and $s(21) = 4$. For updates arriving later than time 21, there is no time for which all five prefixes have appeared in updates, i.e., $s(t) = \infty$. As updates occur closer to the beginning of the table transfer, $s(t)$ decreases steadily until reaching a minimum value at $s(21) = 4$. Note that the table transfer begins at time $t = 21$ and lasts exactly $s(21) = 4$ seconds. After this minimum value, $s(t)$ steadily increases.

In general, we expect a trend of decreasing $s(t)$ as the update under consideration approaches the start of a table transfer and increasing $s(t)$ as we move past the start of a table transfer. Calculated from real data, Figure 3 illustrates the trend of $s(t)$ versus t [2]. Based on this observation, we devise the following basic algorithm to detect table transfers given a stream of updates:

1. For each update, calculate its collection time, $s(t)$.
2. Find all local minima of $s(t)$.
3. Each local minimum is considered as a table transfer. Its time t is the starting time of the transfer, and its collection time $s(t)$ is the duration of the transfer.

If $s(t)$ *monotonically* decreases prior to the beginning of a table transfer and then *monotonically* increases after passing the beginning of a table transfer as shown in Figure 3, then the basic algorithm works perfectly. In BGP data that we have processed, most of the time this is the case. However, sometimes the monotonicity does not hold due to certain patterns of update timing and ordering, or incomplete table transfers. In next section, we introduce some simple tune-ups to adapt our basic algorithm to the vagaries of real BGP data.

3. PRACTICAL TUNE-UPS

The basic Minimum Collection Time approach needs to be adjusted to address a few issues that arise in real BGP data, which is the focus of this section.

[2]The upper limit of 7200 seconds on $s(t)$ is explained in the next section.

3.1 Reducing Computation Load

Incremental updates that are not part of any table transfer can have very long collection times (e.g., on the order of many hours or days). Recall that we are looking for minimum $s(t)$ values and these minimum values correspond to table transfer durations. Computing very large $s(t)$ values requires certain computational cycles but serves very little purpose if we are certain that these values cannot be a minimum.

To reduce computation load, we put an upper bound U on the maximum $s(t)$ value. Once $s(t)$ reaches this upper bound, we are certain the update under consideration cannot be part of a table transfer and we set its $s(t)$ to U. In our implementation, we set $U = 7200$ seconds (i.e., 2 hours). That is, for any $s(t) > 7200$ seconds, we set it to 7200 seconds. Since we are looking for the minimum of $s(t)$, the value of U does not affect the result, as long as it is larger than any whole table transfer duration. It is very unlikely that any table transfer will last up to 2 hours. The use of U is simply a computational convenience. If no safe maximum estimate for a table transfer duration can be inferred, we can simply set U to infinity.

3.2 Expected Table Size

In calculating collection times, the basic approach assumes that the set of prefixes to be announced in the table transfer is known in advance. We can collect this set of prefixes by observing updates sent by the router. A typical router from RIPE or RouteViews announces routes to over 120,000 prefixes. However this is not a static set. For example, suppose the BGP session between the router $R1$ and $R2$ goes down. During the session downtime, $R2$'s route to prefix p is withdrawn by one of its other neighbors. When the session between $R1$ and $R2$ is re-established, $R2$ will send its current routing table to $R1$, but prefix p will not be part of this transfer. Thus, the table transferred after the session re-establishment may not have exactly the same size as the one before the session breakdown. Therefore we should not count *all* the prefixes observed so far when calculating the collection time.

However, we expect that most of the prefixes will still be present in the table transfer. The number of unique prefixes needed to constitute a full table is a parameter in our method, and we denote it as N. If N is too high, we may miss some table transfers; if N is too low, we may falsely classify a surge of incremental updates as a table transfer. Based on our experience with real data, setting N to 99% of the last known table size seems a good engineering choice.

Ideally, we should maintain a routing table while processing the updates, so that we know the table size at any time, and set 99% of it to be the expected table size. As a quick proof of evidence, our current implementation measures the routing table size at the beginning of a month, and uses 99% of this value as the expected table size for the entire month. Although such a quick and dirty implementation seems offering good results as presented in section 4, we are currently developing a simple adaptive algorithm for the routing table size estimation.

3.3 Dealing with Trend Noise

The basic MCT approach assumes that the collection time $s(t)$ decreases *monotonically* prior to the beginning of a table transfer. The collection time then increases *monotonically* after passing the beginning of a table transfer. However, sometimes this monotonicity can be violated. For example, suppose the update stream in Figure 2 is modified slightly as shown in Figure 4. We again assume the full routing table consists of prefixes $p1$, $p2$, $p3$, $p4$ and $p5$. The resulting $s(t)$ values are now $s(10) = 14, s(14) = 10, s(15) = 12, s(21) = 6, s(22) = \infty$. In this case, $s(t)$ still follows a decreasing trend as we approach the table transfer at time 21, but there is a slight increase between the updates at time 14 and 15. Thus the basic MCT approach will find two local minima, one at time 14, and one at time 21, although only the latter corresponds to a real table transfer.

However one may notice that the falsely perceived table transfer at time 14 and the actual table transfer at time 21 have an interesting relation. The falsely perceived table transfer starts at time 14 and ends at time $14 + s(14) = 14 + 10 = 24$ (see Figure 4). This *conflicts* with the second local minimum, which says a table transfer starts at time 21. In other words, a table transfer is starting at the same time when another table transfer is still in progress. Given two local minima $s(t_1)$ and $s(t_2)$ of the collection times, we say they are *conflicting* if $t_1 + s(t_1) > t_2$. In the event of an overlap, the shorter transfer time is taken to be the real table transfer. In our example, there is a conflict between $s(14)$ and $s(21)$. Since $s(21) < s(14)$, we discard $s(14)$ and keep $s(21)$ as the real table transfer.

More formally, assume we have computed $s(t)$ for all updates and have found all local minima in the resulting $s(t)$ values. We then check for and resolve conflicts as follows.

1. Set t_m to the first local minimum.
2. Check all other local minima. If there is any conflicting local minimum $t \mid s(t) < s(t_m)$, discard this local minimum t_m.
3. Otherwise, report that a table transfer starts at t_m and lasts $s(t_m)$ seconds.
4. Set t_m to the next local minimum, repeat step 2, until all local minima have been either reported as table transfers or discarded.

3.4 Multiple Table Transfers

In the event that multiple table transfers occur close to each other in time, as long as one table transfer completes before the second one starts, MCT can still correctly identify each of them. Figure 5 shows an example from real BGP data where four table transfers happened within 35 minutes, and they were identified by four local minima in $s(t)$

However, a table transfer might not necessarily complete before another one starts. For example, suppose the BGP session goes down before an on-going table transfer is completed. When the session is up again, a new table transfer will start. In this case, we may see two local minima in the collection time, $s(t_1)$ for the partial transfer and $s(t_2)$ for the complete transfer. However, these two will conflict with each other, $t_1 + s(t_1) > t_2$, since the partial transfer's collection time must extend into the complete transfer in order to include all prefixes. This is the same characteristic of trend noise, and we handle it with the same technique of choosing the one with shorter collection time as a true table transfer. Thus, MCT can correctly identify complete table transfers even in the presence of multiple partial table transfers close in time.

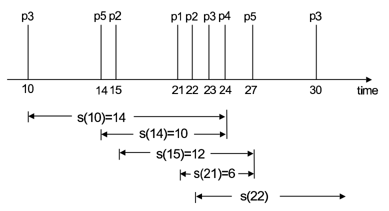

Figure 4: Trend Noise and Conflicting Transfers

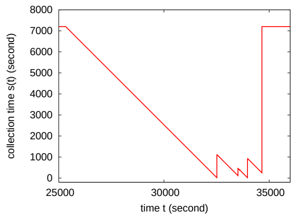

Figure 5: Multiple Table Transfers

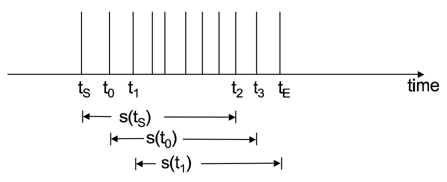

Figure 6: Bottom Searching

3.5 Bottom Searching

MCT assumes that a local minimum $s(t_m)$ (after removing trend noises and partial transfers) corresponds to the start of a full table transfer. As a result of imprecise estimate of expected table size, t_m may not be the exact starting time, and the true starting time could be earlier than t_m. As illustrated in Figure 6, the table transfer starts at t_S and ends at t_E. But $s(t_S)$ ends earlier at t_2, because we have already seen 99% of the table. A similar early end occurs for $s(t_0)$ and $s(t_1)$. These collection times, $s(t_S)$, $s(t_0)$, and $s(t_1)$, will have similar values, and all appear at the bottom of the valley in $s(t) \sim t$ plot. Depending on the timing of updates, any one of the three can have the minimum value, but only $s(t_S)$ is the true start of the table transfer.

To accommodate this situation, we apply a ***bottom searching threshold***, B. After finding the minimum collection time $s(t_m)$, we look *back* B seconds in updates starting at t_m and pick the earliest update between $t_m - B$ and t_m. From our experience of real data, we observed that t_S is usually only a few seconds before t_m when t_S itself is not the detected local minimum. When t_S is the actual detected local minimum, our bottom searching would not find a false start, since right before the table transfer there must be a relatively large time gap with no update in it (i.e., session downtime plus session re-establishment time). From our experiments, we found that setting $B = 10$ seconds is effective in locating the true start of a table transfer.

3.6 Summary of the Algorithm

After applying the tune-ups mentioned above, the final MCT algorithm can be summarized as follows.

1. Calculate collection time $s(t)$ for all updates. Use $U = 7200$ seconds as the upper limit of $s(t)$, and use $N = 99\%$ of the last known table size as the expected table size.

2. Find all local minima of $s(t)$.

3. Resolve conflicts, which can be caused by trend noises or incomplete table transfers.

4. For each local minimum, search for the true start of the table transfer using bottom searching threshold $B = 10$ seconds.

4. EVALUATION

We now evaluate the MCT algorithm using BGP log data. We verify the detection accuracy with BGP session state messages in RIPE data and apply MCT to RouteViews data.

4.1 Verification with Session State Messages

The RIPE RRC00 monitor (located at Amsterdam) logs BGP session state messages along with regular BGP updates. From session state messages, we can infer BGP session breakdown and re-establishment. Since a session reset is supposed to trigger a table transfer, we can use MCT to identify table transfers, and verify the results with session establishment messages. Our validation uses three months of RIPE RRC00 data, from January 2002 to March 2002, which includes update streams from 14 different peers. This date range was chosen because RIPE disabled its monitor's BGP KeepAlive timer on October 17, 2002. Prior to October 2002, the HoldTime was set to 60 seconds and we believe this contributed to BGP session resets from time to time, which is better suited for our verification purpose.

During the three months, we detected 495 cases that had a full table transfer and had a corresponding session establishment message indicating a reset. There were 29 cases that had a session establishment message but no table transfer was detected, and 15 cases where MCT detected a table transfer but found no session establishment message. Figure 7 shows the counts over different peers. Overall most session resets (94%) cause full table transfers, and most table transfers (97%) are triggered by session resets.

The 29 resets with no detected table transfers are due to the fact that, not all session resets lead to *complete* table transfers. Figure 8 shows one such example from real BGP data. Four consecutive session resets were followed by 33681, 65, 148, and 107133 announced prefixes. In the first three cases, the table transfer could not complete since the session went down again. Only the last one completed a table transfer, which is correctly identified by our method.

The 15 table transfers without corresponding session establishment messages could be due to missing session state messages or "soft reset". Though session state messages are generated locally by the monitor, its logging may not be precise due to the presence of a software bug [3]. In our data, we found evidence of missing session state messages. For example, one session went through two consecutive session establishment processes without a session breakdown message in between. "Soft reset" [1] can be used by router operators to signal changes after router reconfiguration. After a change such as a change in routing policy, the router may reannounce the entire table without resetting the BGP session. Finally, it is possible that the BGP peer may suffer from connection stability issues with all its upstream peers and, after some unfortunate failures, the router may lose and then re-learn (and hence re-announce) almost all of its routing table.

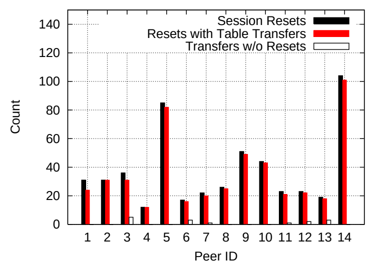

Figure 7: Session Resets and Table Transfers

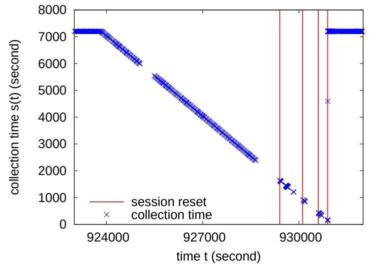

Figure 8: Multiple Incomplete Table Transfers

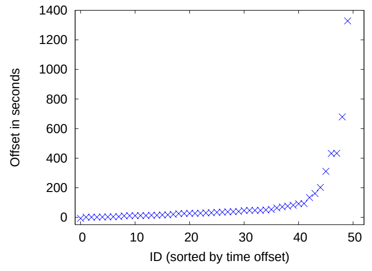

Figure 9: Non-zero Time Offset

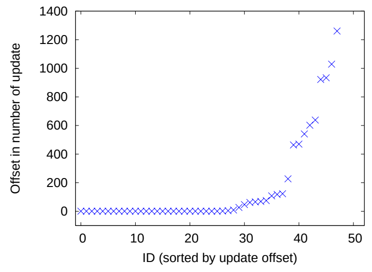

Figure 10: Non-zero update offset

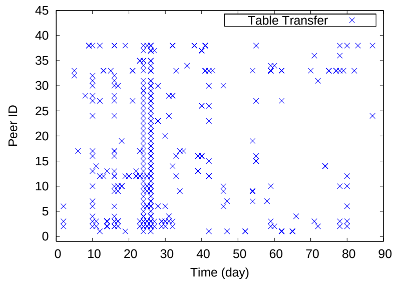

Figure 11: Table Transfers in RouteViews Data

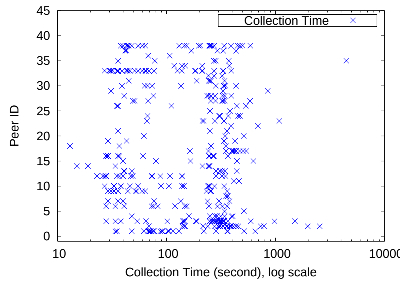

Figure 12: Collection Times in RouteViews Data

We quantify the detection accuracy for the 495 cases that have both resets recorded and table transfer detected. We take the *first routing update message* after session establishment as the real start of a table transfer, as opposed to the the session establishment message itself. We then compare this real start with the one our method finds. The difference between these two is called "offset," and is measured in terms of number of seconds and number of updates. Out of the 495 cases, 445 (90%) cases have offsets of *zero* second, i.e., our method finds the exact starting point of the table transfer with no error.

For the 50 cases with non-zero offset, Figure 9 shows their time offset and Figure 10 shows the number of updates within this time offset. (Note that the same ID in the two figures does not correspond to the same case.) For most of the cases the offset is small. For example, in 24 cases, MCT misses the real start by less than 30 seconds, and in 27 cases we miss the real start by only *one* update. We found a few cases to have large offsets. The largest time offset is 1328 seconds but it has only 1 update within this period, which means there is a large gap between the first update and the second update of the table transfer, and our method only misses a single update. There are two extreme cases with very large number of updates, 55k and 15k, respectively (not plotted in Figure 10 to make it easier to read). After careful inspection we found evidence of imprecise session state logs and we suspect that there were in fact two quick session resets and a missing state message for the second reset.

Overall, MCT detected 100% full routing table transfers triggered by session resets, and in 90% of the cases it pinpointed the exact starting time of table transfers with no error.

4.2 Application to RouteViews Data

The RouteViews project has collected valuable BGP data for a number of years, however the routing updates do not contain session state messages. MCT provides the first practical way to accurately identify table transfers in RouteViews' data. We applied MCT on three recent months of data starting January 2005 to March 2005 collected from the Oregon collector of RouteViews. Figure 11 shows when table transfers are detected for each RouteViews peer. There are totally 362 table transfers in these three months for 37 peers. It is interesting to notice that on days 24 and 26 almost all peers have table transfers, suggesting that RouteViews monitor may have experienced problems at that time. Figure 12 shows the collection times of every table transfer for each peer. Most of the collection times fall in either 20 to 70 seconds, or 200 to 500 seconds, with a few greater than 1000 seconds. Thus most table transfers finish within 10 minutes. For each individual peer, Figure 13 shows the number of table transfers, and Figure 14 shows the percentage of table transfer updates in the total number of updates.

5. RELATED WORK

Prior work that used BGP updates has highlighted the need to clean the BGP data to differentiate table transfers from incremental updates. [8] uses session state messages in BGP logs to identify the start of a session re-establishment and thus the beginning of a table transfer. However, RouteViews logs, which have many years of valuable data, don't contain such state messages.

The method employed in [5] does not explicitly identify table transfers, but instead removes all duplicate announcements from the update stream. A reset of BGP session trig-

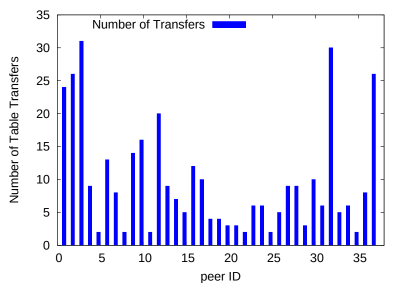

Figure 13: Number of Table Transfers

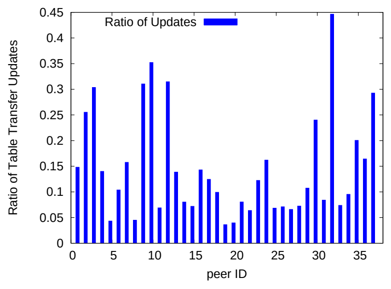

Figure 14: Ratio of Table Transfer Updates

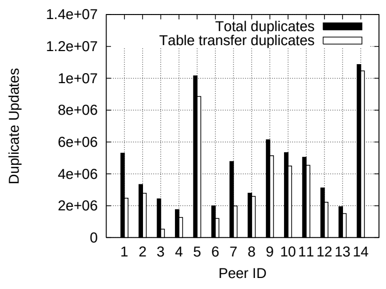

Figure 15: Removing Duplicate Updates

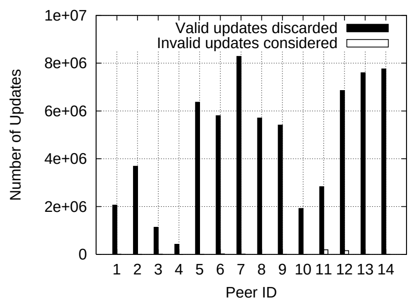

Figure 16: Bin-based Discard

gers a complete table transfer and since the session downtime is usually short compared with routing changes, BGP updates sent after session re-establishment should consist primarily of duplicate announcements. However, with our three-month RIPE data set, Figure 15 shows that duplicate announcements are not produced only by table transfers. Eliminating all duplicate announcements removes both table transfer updates and updates due to other factors, which could be useful in studying routing dynamics. Although this shortcoming does not affect the result in [5], it limits the applicability of this method as a general approach to deal with table transfers in BGP logs.

The method in [2] makes use of a general observation that updates due to table transfers occur in bursts. This method splits the update stream into 30-second bins and discards any bin that contains more than 1000 prefixes, regarding them as part of table transfers. In Figure 16, we plot the number of valid updates that this method discards, and table transfer updates this method misses using the three-month RIPE data set. It is clear that this method doesn't miss many table transfer updates, but it falsely discards a large number of legitimate updates.

[4] presents a bin-based heuristic to detect session resets in the Internet, not limited to BGP sessions between the monitor and its peers. [9] presents a heuristic to detect session resets between a network's border routers and their external peers. Their scheme takes into account a majority of routes shifting from one neighbor to another, in a small interval of time as an indication of session reset or restoration. Both these approaches present heuristics for inferring session resets, but do not directly address all the issues arising with session resets between monitoring points and its direct peers.

6. CONCLUSION

We have developed a Minimum Collection Time (MCT) algorithm that accurately identify the start and duration of BGP routing table transfers triggered by peering session resets from a stream of routing updates. Using three months of BGP data from 14 different peers, MCT can identify 100% of full routing table transfers triggered by session resets, and in 90% of the cases we were able to pinpoint the exact start without any error. MCT is particularly useful in processing RouteViews BGP update logs which do not contain session state messages.

Built on the success in detecting session resets with direct peers, we are extending the basic idea to detect *partial* table transfers from update streams. A partial table transfer can be the result of a full table transfer triggered by BGP session resets that are one or more hops away from the monitor. Being able to inferring remote session resets is interesting since it will help us better understand BGP routing dynamics in the operational Internet.

7. REFERENCES

[1] Cisco documentation: Configuring BGP, 2003.

[2] D. Andersen, N. Feamster, S. Bauer, and H. Balakrishnan. Topology inference from bgp routing dynamics. In *ACM SIGCOMM Internet Measurement Workshop (IMW)*, 2002.

[3] H. Kong. The consistency verification of Zebra BGP data collection. Technical report, Agilent Labs, China, 2003.

[4] O. Maennel and A. Feldmann. Realistic BGP traffic for test labs. In *Proc. of ACM SIGCOMM*, 2002.

[5] J. Rexford, J. Wang, Z. Xiao, and Y. Zhang. BGP routing stability of popular destinations. In *ACM SIGCOMM Internet Measurement Workshop (IMW)*, 2002.

[6] RIPE Routing Information Service. `http://www.ripe.net/projects/ris/`.

[7] The RouteViews project. `http://www.routeviews.org/`.

[8] L. Wang, X. Zhao, D. Pei, R. Bush, D. Massey, A. Mankin, S. F. Wu, and L. Zhang. Observation and analysis of BGP behavior under stress. In *ACM SIGCOMM Internet Measurement Workshop (IMW)*, 2002.

[9] J. Wu, Z. M. Mao, J. Rexford, and J. Wang. Finding a needle in a haystack: Pinpointing significant BGP routing changes in an IP network. In *Symposium on Networked System Design and Implementation (NSDI)*, May 2005.