Increasing the Coverage of a Cooperative Internet Topology Discovery Algorithm

Benoit Donnet[1], Bradley Huffaker[2], Timur Friedman[3], and K.C. Claffy[2]

[1] Université Catholique de Louvain – CSE Department, Belgium
[2] Caida – San Diego Supercomputer Center, USA
[3] Université Pierre & Marie Curie – Laboratoire LIP6/CNRS, France

Abstract. Recently, Doubletree, a cooperative algorithm for large-scale topology discovery at the IP level, was introduced. Compared to classic probing systems, Doubletree discovers almost as many nodes and links while strongly reducing the quantity of probes sent. This paper examines the problem of the nodes and links missed by Doubletree. In particular, this paper's first contribution is to carefully describe properties of the nodes and links that Doubletree fails to discover. We explain incomplete coverage as a consequence of the way Doubletree models the network: a tree-like structure of routes. But routes do not strictly form trees, due to load balancing and routing changes. This paper's second contribution is the Windowed Doubletree algorithm, which increases Doubletree's coverage up to 16% without increasing its load. Compared to classic Doubletree, Windowed Doubletree does not start probing at a fixed hop distance from each monitor, but randomly picks a value from a range of possible values.

1 Introduction

Today's most extensive tracing system at the IP interface level, *skitter* [1], uses 24 monitors, each targeting on the order of one million destinations. In the fashion of skitter, *scamper* [2] makes use of several monitors to traceroute IPv6 networks. Other well known systems, such as Ripe NCC's *TTM* [3] and Nlanr's *AMP* [4], employ a larger set of monitors, on the order of one- to two-hundred, but they avoid probing outside their own network. Recent work indicates, however, the need to increase the number of traceroute sources in order to obtain a more accurate topology measurement. Indeed, it has been shown that reliance upon a relatively small number of monitors to generate a graph of the internet can introduce unwanted biases [5], [6].

One way of rapidly creating a large distributed monitoring infrastructure would be to deploy traceroute monitors in an easily downloadable and readily usable piece of software, such as a screensaver. This was first proposed by Jörg Nonnenmacher, as reported by Cheswick et al. [7]. The first publicly downloadable distributed route tracing tool is Dimes [8], released as a daemon in September 2004.

However, building such a large infrastructure leads to potential scaling issues: the quantity of probes launched might consume large amounts of network

I.F. Akyildiz et al. (Eds.): NETWORKING 2007, LNCS 4479, pp. 738–748, 2007.
© IFIP International Federation for Information Processing 2007

resources and the probes sent from many vantage points might appear to end-hosts or firewalls as a distributed attack. These problems were quantified in our prior work [9].

The Doubletree algorithm [9] is designed to perform large-scale topology discovery in a network friendly manner. Doubletree avoids retracing the same routes in the internet by taking advantage of the tree-like structure of routes fanning out from a source or converging at a destination. The key to Doubletree is that the monitors share information regarding the paths that they have explored. If one monitor has already probed a given path to a destination then another monitor should avoid that path. Probing in this manner can significantly reduce the load on routers and destinations while maintaining high node and link coverage.

However, even if Doubletree's results, in terms of links and nodes coverage, are high (above 90%, compared to classic probing, such as skitter), some nodes and links are not reachable by Doubletree. This paper's first contribution is a careful study of the topological data missed by Doubletree. Based on a subset of skitter data, we simulate Doubletree and show that the majority of nodes and links missed is located between 9 and 20 hops from Doubletree monitors. We believe that these missed links and nodes are located between routes' divergence and convergence points in the network. Doubletree does not take into account such points due to the way it models the network. Considering the tree-like structure of routes implies a static view of the network. But convergence and divergence of routes arise because of network dynamics as such points are created by load balancing or routing changes. Load balancing refers to the fact that routers might spread their traffic across multiple paths [10]. Three policies might be considered: per-flow, per-packet and per-destination. Note that impacts of load balancing on traceroute-like probing are detailed in [11].

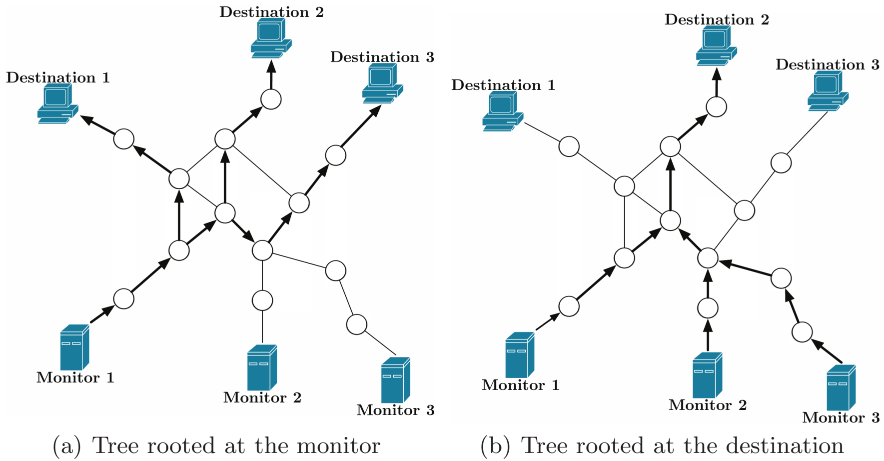
Based on this knowledge of missed information, we propose an improvement to Doubletree called *Windowed Doubletree*, this paper's second contribution. Instead of starting probing at a fixed point somewhere in the network, Windowed Doubletree builds a window based on the location of nodes and links missed. For each destination to probe, a Windowed Doubletree monitor picks up randomly a value in this window and then, probes in the same way as classic Doubletree.

We evaluate Windowed Doubletree and find that it is able to increase the classic Doubletree coverage by 16% while maintaining nearly the same impact on destinations and router interfaces as classic Doubletree.

The remainder of this paper is organized as follows: Sec. 2 presents Doubletree; Sec. 3 discusses Doubletree's limitations; Sec. 4 introduces and evaluates Windowed Doubletree, our improvement to Doubletree based on knowledge of missed information; finally, Sec. 5 concludes this paper by summarizing its main contributions.

2 Doubletree

Doubletree [9] takes advantage of the tree-like structure of routes in the context of probing, as illustrated in Fig. 1. Routes leading out from a monitor towards multiple destinations form a tree-like structure rooted at the monitor (Fig. 1(a)).



(a) Tree rooted at the monitor (b) Tree rooted at the destination

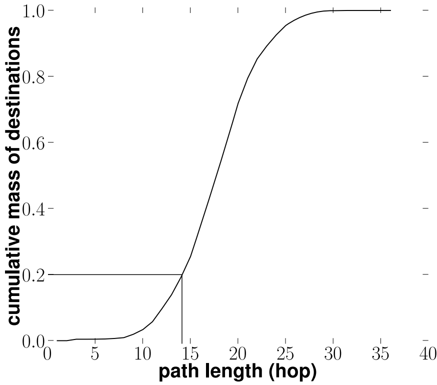
Fig. 1. Tree-like structure of routes

Similarly, routes converging towards a destination from multiple monitors form a tree-like structure, but rooted at the destination (Fig. 1(b)). A monitor probes hop by hop so long as it encounters previously unknown interfaces. However, once it encounters a known interface, it stops, assuming that it has touched a tree and the rest of the path to the root is also known. Using these trees suggests two different probing schemes: *backwards* (monitor-rooted tree – decreasing TTLs) and *forwards* (destination-rooted tree – increasing TTLs).

For both backwards and forwards probing, Doubletree uses stop sets. The one for backwards probing, called the *local stop set*, consists of all interfaces already seen by that monitor. Forwards probing uses the *global stop set* of (interface, destination) pairs accumulated from all monitors. A pair enters the global stop set if a monitor receives a packet from the interface in reply to a probe sent towards the destination address.

A Doubletree monitor starts probing for a destination at some number of hops h from itself. It will probe forwards at $h+1$, $h+2$, etc., adding to the global stop set at each hop, until it encounters either the destination or a member of the global stop set. It will then probe backwards at $h-1$, $h-2$, etc., adding to both the local and global stop sets at each hop, until it either has reached the distance of one hop or it encounters a member of the local stop set. It then proceeds to probe for the next destination. When it has completed probing for all destinations, the global stop set is communicated to the next monitor. Note that in the special case where there is no response at distance h, the distance is halved, and halved again until there is a reply, and probing continues forwards and backwards from that point.

Doubletree has a single tunable parameter, the initial hop distance h. While Doubletree largely limits redundancy on destinations once hop-by-hop probing is underway, its global stop set cannot prevent the initial probe from reaching a destination if h is set too high. Therefore, each monitor sets its own value for h in



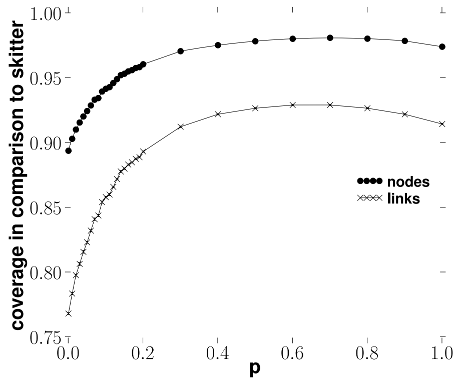


Fig. 2. Cumulative mass plot of path lengths from skitter monitor `champagne`

Fig. 3. Doubletree coverage compared to classic probing

terms of the probability p that a probe sent h hops towards a randomly selected destination will actually hit that destination. Fig. 2 shows the cumulative mass function for this probability for skitter monitor `champagne`. If one considers as reasonable a 0.2 probability of hitting a responding destination on the first probe, the `champagne` monitor must choose $h \leq 14$.

3 Doubletree Limitations

3.1 Methodology

Our study of Doubletree's limitations is based on skitter data from August 1st through 3rd, 2004. At this time, skitter was deployed on 24 monitors scattered around the world: the United States, Canada, the United Kingdom, France, Sweden, the Netherlands, Japan and New Zealand. The different monitors shared a common destination set of 971,080 IPv4 addresses. Each monitor cycled through the destination set at its own rate, taking typically three days to complete a cycle. For the purpose of our studies, in order to reduce computing time to a manageable level, we worked from a limited set of 50,000 destinations, randomly chosen from the original set.

We conducted simulations based on the skitter data, assuming that each of the 24 skitter monitors applied Doubletree, as described in Sec. 2. We implemented the same communication scheme for Doubletree as the one described by Donnet et al. [9, Sec. IV.B.]. Essentially, a random order was chosen for the monitors and each one simulated the running of Doubletree in turn. Each monitor added to the global set the $(\text{interface}, \text{destination})$ pairs that it encountered, and passed the set to the subsequent monitor.

A single experiment used traceroutes from all 24 monitors to a common set of 50,000 destinations chosen at random. Each data point in plots represents the mean value over fifteen runs of the experiment, each run using a different set

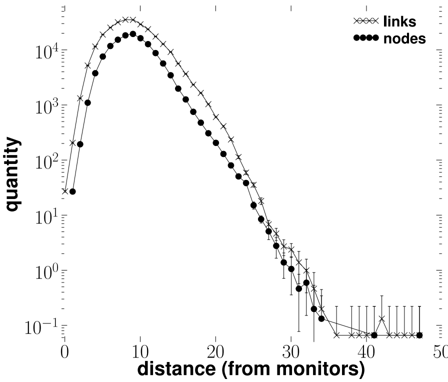


Fig. 4. Nodes and links at each hop

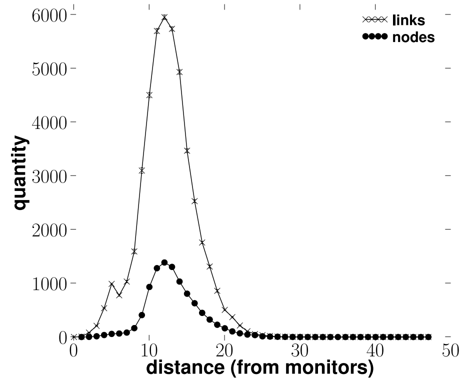


Fig. 5. Nodes and links missed - $p = 0.05$

of 50,000 destinations generated at random. No destination was used more than once over the fifteen runs. We determined 95% confidence intervals for the mean based, since the sample size was relatively small, on the Student t distribution. These intervals are typically, though not in all cases, too tight to appear on the plots.

3.2 Results

Fig. 3 shows the mean Doubletree coverage (nodes and links) compared to classic probing for some p values (an increment of 0.01 between 0 and 0.2 and an increment of 0.1 between 0.2 and 1). A coverage value of 1 would indicate that Doubletree is able to discover the same proportion of nodes and links as classic probing. A p value of zero means that a Doubletree monitor performs forwards probing only. At the other extreme, a p value of 1 indicates that a Doubletree monitor, in all cases when a destination replies to the first probe, performs backwards probing only.

We see that the coverage increases with p but never reaches 1. A coverage peak is reached when $p = 0.8$. After that point, the coverage decreases a little bit. As we can see, though it can achieve over 90% coverage, Doubletree, in its basic form, has room for improvement.

Fig. 4 shows the average number, over the fifteen destination subsets, of nodes and links discovered at each hop by skitter. The vertical axis, in log-scale, gives the quantity and the horizontal axis the hop count. As a given node might be located at a distance x from one monitor and at a distance y from another (the same problem might occur with links), we sort all nodes and links by distance and plot, for each node or link, the minimum distance. This methodology gives thus a monitor-independent representation. Further, note that for the links, the distance plotted is the distance of the start of the link. Finally, one can see that Fig. 4 shows quantities below 10^0 for some nodes and links. This is a consequence

of the mean over the fifteen destination subsets. Only a few nodes (or links) might appear in a few subsets at some distances from the monitors.

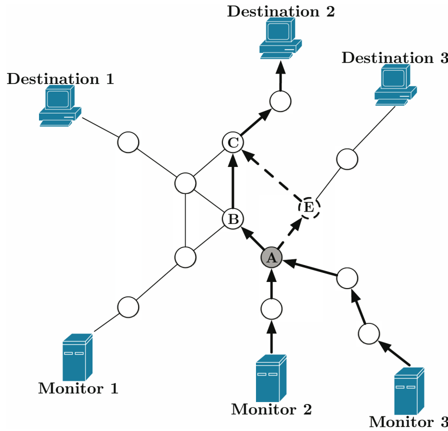
In our dataset, skitter discovers, on average, 131,780 nodes and 279,799 links. Looking at Fig. 4, we note that there is a rapid increase in the number of nodes and links at each hop until reaching a peak close to the traceroute sources: at hop 8 corresponding to 35,620 different links and at hop 9 corresponding to 19,662 nodes. After this peak, the number of nodes and links per hop decreases slowly until reaching a minimum (or a value very close to the minimum) after the 35th hop.

Fig. 5 investigates the nodes and links missed with Doubletree when $p = 0.05$. This is in the range of p values recommended by our prior work [9, Sec. IV.B.], as it provides a good compromise between redundancy reduction and coverage. Fig. 5 plots the quantity of nodes and links missed at each hop.

We see that the majority of nodes and links missed are located between 9 and 20 hops from a monitor, where the majority of nodes and links are located, according to Fig. 4. A peak (5,953 links and 1,387 nodes are not elicited) is reached at 12 hops from the sources. After 20 hops, the amount of topological information missed becomes negligible.

We believe that this information missed is located between routes' divergence and convergence points. Doubletree encounters difficulties in discovering nodes and links within such points because of its stopping rules. For the sake of explanation, let us consider the destination-rooted tree (see Fig. 6). Suppose that *Monitor 2* has $h \leq 4$ and probes *Destination 2*. It will, as explained in Sec. 2, probe forwards from h and backwards from $h-1$. It will also populate the global stop set with the $(\text{interface}, \text{destination})$ pairs it encounters. When *Monitor 2* has finished probing, it sends its global stop set to *Monitor 3*. Suppose that *Monitor 3* has $h \leq 2$. When discovering the path to *Destination 2*, it will probe forwards until reaching the gray interface (A). As this interface towards *Destination 2* was previously discovered by another monitor in the system, *Monitor 3* stops probing, considering that the rest of the path to the root of the tree is already known. However, because of load balancing or routing change, the path from the gray interface towards *Destination 2* has changed. As a consequence, in this example, one node (E) and two links (dashed lines - $A \rightarrow E$ and $E \rightarrow C$) will not be discovered. The same reasoning applies for the monitor-rooted tree but with backwards probing and a stopping rule based on the local stop set.

This case occurs because of the way Doubletree models the network. As explained in Sec. 2, Doubletree assumes, in the context of probing, that the routes have a tree-like structure. This is true in a large proportion as suggested by Doubletree's coverage results (see Fig. 3), but this hypothesis implies a static view of the network. When a Doubletree monitor stops probing towards the root of a tree, it assumes that the rest of the path to the tree is both known and unchanged since earlier probing. The existence of routes' convergence and divergence points, however, imply a dynamic view of the network, as some parts of the network might change due to load balancing and routing.



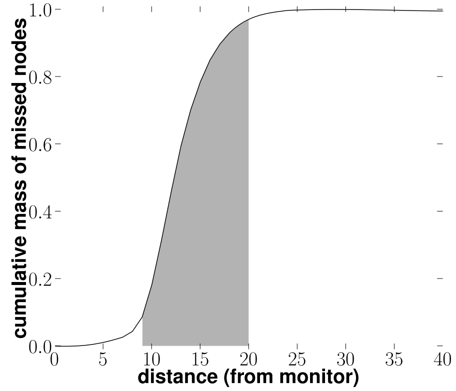


Fig. 6. Route's convergence point in the case of destination rooted tree

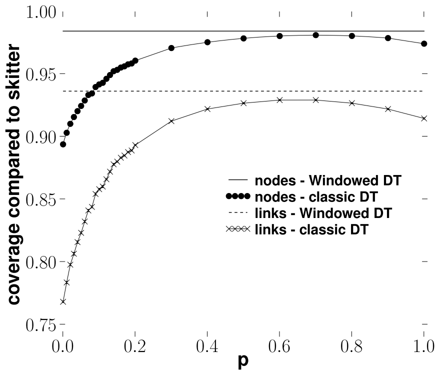
Fig. 7. Windowed Doubletree - the `champagne` monitor

4 Windowed Doubletree

As explained in Sec. 3, Doubletree will miss node E (see Fig. 6) if it first probes path $A \rightarrow B \rightarrow C$ and then $A \rightarrow E \rightarrow C$. E will be hidden behind the shared interfaces of A and C. That is unless Doubletree starts its probing on the second hop in the paths between A and C. This will allow it to discover both B and E. However, as explained in Sec. 2, Doubletree, in its classic form, starts probing from a fixed value of h for all destinations. Consequently, unless the inconsistency between the paths occurs at its fixed h, it will not be discovered.

Our idea is to randomize the distance at which probing will start. Rather than launching the first probe at a constant value of h, each monitor will randomly pick a value of h in the window of missing nodes and links. This is illustrated in Fig. 7. The horizontal axis gives the distance to monitors of missed nodes and the vertical axis the cumulative mass of missed nodes. The window, between 9 and 20 hops, is shown by the shaded area. Note that taking into account smaller values for the window would raise the risk of *intra-monitor redundancy* [9, Sec. III.A.], i.e., the duplication of a monitor's own work, that leads to inefficiency. Introducing this element of randomness in probing dramatically increases the probability that at least one Doubletree trace will start probing inside a route's divergence and convergence segment. Thus allowing Doubletree to discover the nodes and links hidden within this segments.

The rest of Doubletree's behavior is left unchanged. Probing continues forwards from h and backwards from $h-1$ while using global and local stop set to decide when probing must stop. Finally, all monitors continues to cooperate in order to exchange their global stop set. We call this improved algorithm *Windowed Doubletree*.



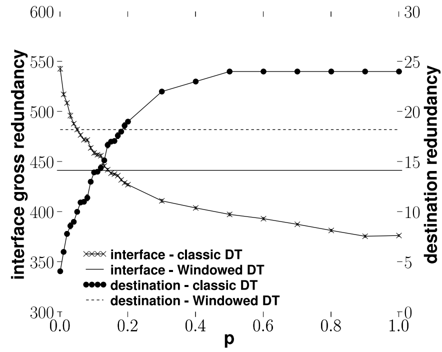


Fig. 8. Coverage in comparison to classic Doubletree and classic probing

Fig. 9. Windowed Doubletree redundancy (95th percentile)

4.1 Evaluation

Our study was based on the same data set as the one described in Sec. 3.1. We assumed that Windowed Doubletree is running, as described in Sec. 4, on the skitter monitors, during the same period of time that the skitter data represents. In addition, we simulated randomness to pick a value for h within the window with the Mersenne Twister MT19937 pseudorandom number generator [12].

The main performance metric for a probing system is the extent to which it discovers what it should. Fig. 8 shows the Windowed Doubletree node and link coverage compared to classic probing and classic Doubletree. The horizontal axis gives the probability p (i.e., the probability of hitting a destination with the first probe sent) between 0 and 1. The vertical axis gives the coverage proportion in comparison to classic probing, i.e., skitter. A value of 1 would mean that Doubletree is able to discover the same proportion of nodes and links as classic probing. We compare Windowed Doubletree (the horizontal lines) with classic Doubletree (the curves).

Looking first at the node coverage (upper curve), we see that Windowed Doubletree (plain line) is able to discover more nodes than classic Doubletree. The increase is slight: between 9% ($p = 0$) and 0.3% ($p = 0.8$). Therefore, reaching nearly the same proportion of nodes discovered by using classic Doubletree would mean considering a p value of 0.8. Previous work [9] pointed out that such a value for the parameter p is not advisable due to the load on destinations. Indeed, a value of $p = 0.8$ means that in 80% of the cases, the first probe sent by a Doubletree monitor will hit a destination, which can be interpreted by final hosts and firewalls as an attack. With link coverage, the difference is much greater: Windowed Doubletree captures between 16% (p=0) and 0.7% (p=0.8) more links than classic Doubletree.

Windowed Doubletree is thus able to discover more than 98% of the nodes and 93% of the links discovered in classic probing. Windowed Doubletree increases the coverage of classic Doubletree.

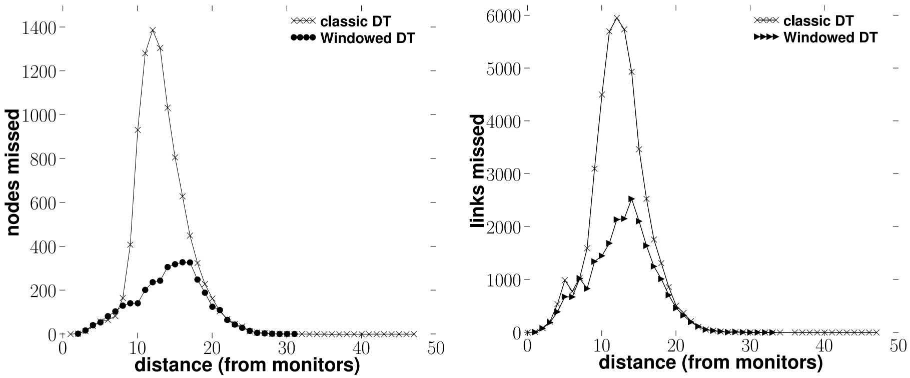


Fig. 10. Nodes missed with Windowed Doubletree compared to classic Doubletree

Fig. 11. Links missed with Windowed Doubletree compared to classic Doubletree

The goal of applying Doubletree is to reduce the load on network interfaces in routers and, more importantly, at destinations. If Windowed Doubletree increased this load, it would be a concern.

Fig. 9 shows the redundancy for routers interfaces (left vertical axis) and for destinations (right vertical axis). With regard to router interface redundancy, we are concerned by the overall load. We therefore count the total number of visits to an interface. We call this metric *gross redundancy*. We evaluate the destination redundancy by counting the number of monitors that hit a given destination. The maximum value is thus the total number of monitors in the system, i.e., 24 in our case. For both destination and router interfaces, we are concerned with the extreme values, so we plot the 95$^{\text{th}}$ percentile.

Looking first at the gross redundancy, we see that Windowed Doubletree produces the same amount of redundancy than the classic Doubletree with $p = 0.14$. This corresponds to a value within the range of p values advised by previous work [9, Sec. IV.B.]. Note that the 95$^{\text{th}}$ percentile for the internal interface gross redundancy using a skitter-like approach is 1340 (not shown on Fig. 9). If we compare Windowed Doubletree to the skitter-like approach, Windowed Doubletree allows a reduction in redundancy of 67.09%.

The destination redundancy caused by Windowed Doubletree corresponds to that produced by classic Doubletree with $p = 0.18$ (i.e., 18), which belongs to the advised range of p values. The small increase in destination redundancy is due to the fact that the choice of the h value with Windowed Doubletree is no longer trying to minimize the risk of hitting a destination with the first probe sent. However, Fig. 9 shows that randomly picking a value within a range of values does not lead to disastrous destination redundancy. Finally, note that classic probing generates a destination redundancy of 24 (not shown in Fig. 9), i.e., the number of monitors in our simulations.

Fig. 10 and 11 compare classic Doubletree with Windowed Doubletree in terms of nodes and links missed. The vertical axis gives the quantity of information

(nodes or links) missed and the horizontal axis the distance (in term of number of hops) from the monitors. In the fashion of Fig. 4, nodes and links are sorted by distance and for each node and link, we plot the minimum distance.

Obviously, as suggested by Fig. 8, by introducing an element of randomness in probing, we are able to reduce the quantity of nodes and links missed by classic Doubletree. Looking first at the nodes (Fig. 10), we see that the maximum quantity of nodes missed is located further from monitors: 16 hops (328 nodes) instead of 12 hops (1387 nodes). The same phenomenon appears for links: the maximum quantity of links missed is located 14 hops (2,528 links) from monitors instead of 12 hops (5,953 links). We finally notice that, unlike classic Doubletree, Windowed Doubletree is able to discover nodes and links that are located beyond 30 hops from monitors.

5 Conclusion

In this paper, we improved a cooperative topology discovery algorithm, Doubletree, in order to increase its node and link coverage. We first studied the properties of Doubletree losses and found that these losses are due to the hypothesis Doubletree makes on how the routes are modeled when probing. We also determined that the nodes and links missed are located between routes' divergence and convergence points.

Based on knowledge of nodes and links missed, we proposed Windowed Doubletree. Instead of starting probing at a fixed point in the network as classic Doubletree does, Windowed Doubletree builds a window based on the location of nodes and links missed. For each destination to probe, Windowed Doubletree randomly picks up a value in this window and starts probing from this point. We demonstrated that, by introducing an element of randomness in probing, Windowed Doubletree is able to discover more nodes and links than classic Doubletree while maintaining a low impact on routers and destinations.

Acknowledgements

Mr. Donnet's work was partially supported by the European Commission-funded 034819 OneLab project and by an internship at Caida.

References

1. Huffaker, B., Plummer, D., Moore, D., claffy, k.: Topology discovery by active probing. In: Proc. Symposium on Applications and the Internet. (2002)
2. Luckie, M.: (IPv6 scamper) WAND Network Research Group.
3. Georgatos, F., Gruber, F., Karrenberg, D., Santcroos, M., Susanj, A., Uijterwaal, H., Wilhelm, R.: Providing active measurements as a regular service for ISPs. In: Proc. Passive and Active Measurement (PAM) Workshop. (2001)
4. McGregor, A., Braun, H.W., Brown, J.: The NLANR network analysis infrastructure. IEEE Communications Magazine **38**(5) (2000)

5. Lakhina, A., Byers, J., Crovella, M., Xie, P.: Sampling biases in IP topology measurements. In: Proc. IEEE INFOCOM. (2003)
6. Clauset, A., Moore, C.: Traceroute sampling makes random graphs appear to have power law degree distributions. cond-mat 0312674, arXiv (2004)
7. Cheswick, B., Burch, H., Branigan, S.: Mapping and visualizing the internet. In: Proc. USENIX Annual Technical Conference. (2000)
8. Shavitt, Y., Shir, E.: DIMES: Let the internet measure itself. ACM SIGCOMM Computer Communication Review **35**(5) (2005)
9. Donnet, B., Raoult, P., Friedman, T., Crovella, M.: Deployment of an algorithm for large-scale topology discovery. IEEE Journal on Selected Areas in Communications, Sampling the Internet: Techniques and Applications **24**(12) (2006) 2210–2220
10. Thaler, D., Hopps, C.: Multipath issues in unicast and multicast next-hop selection. RFC 2991, Internet Engineering Task Force (2000)
11. Augustin, B., Cuvellier, X., Orgogozo, B., Viger, F., T., F., Latapy, M., Magnien, C., Teixeira, R.: Avoiding traceroute anomalies with Paris traceroute. In: Proc. Internet Measurement Conference (IMC). (2006)
12. Matsumoto, M., Nishimura, T.: Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. ACM Transactions on Modeling and Computer Simulation **8**(1) (1998) 3–30