Improving the Reliability of Internet Paths with One-hop Source Routing

Krishna P. Gummadi, Harsha V. Madhyastha,
Steven D. Gribble, Henry M. Levy, and David Wetherall
Department of Computer Science & Engineering
University of Washington
{gummadi, harsha, gribble, levy, djw}@cs.washington.edu

Abstract

Recent work has focused on increasing availability in the face of Internet path failures. To date, proposed solutions have relied on complex routing and path-monitoring schemes, trading scalability for availability among a relatively small set of hosts.

This paper proposes a simple, scalable approach to recover from Internet path failures. Our contributions are threefold. First, we conduct a broad measurement study of Internet path failures on a collection of 3,153 Internet destinations consisting of popular Web servers, broadband hosts, and randomly selected nodes. We monitored these destinations from 67 PlanetLab vantage points over a period of seven days, and found availabilities ranging from 99.6% for servers to 94.4% for broadband hosts. When failures do occur, many appear too close to the destination (e.g., last-hop and end-host failures) to be mitigated through alternative routing techniques of any kind. Second, we show that for the failures that *can* be addressed through routing, a simple, scalable technique, called *one-hop source routing*, can achieve close to the maximum benefit available with very low overhead. When a path failure occurs, our scheme attempts to recover from it by routing *indirectly* through a small set of *randomly chosen* intermediaries.

Third, we implemented and deployed a prototype one-hop source routing infrastructure on PlanetLab. Over a three day period, we repeatedly fetched documents from 982 popular Internet Web servers and used one-hop source routing to attempt to route around the failures we observed. Our results show that our prototype successfully recovered from 56% of network failures. However, we also found a large number of server failures that cannot be addressed through alternative routing.

Our research demonstrates that one-hop source routing is easy to implement, adds negligible overhead, and achieves close to the maximum benefit available to indirect routing schemes, *without* the need for path monitoring, history, or a-priori knowledge of any kind.

1 Introduction

Internet reliability demands continue to escalate as the Internet evolves to support applications such as banking and telephony. Yet studies over the past decade have consistently shown that the reliability of Internet paths falls far short of the "five 9s" (99.999%) of availability expected in the public-switched telephone network [11]. Small-scale studies performed in 1994 and 2000 found the chance of encountering a major routing pathology along a path to be 1.5% to 3.3% [17, 26].

Previous research has attempted to improve Internet reliability by various means, including server replication, multi-homing, or overlay networks. While effective, each of these techniques has limitations. For example, replication through clustering or content-delivery networks is expensive and commonly limited to high-end Web sites. Multi-homing (provisioning a site with multiple ISP links) protects against single-link failure, but it cannot avoid the long BGP fail-over times required to switch away from a bad path [12]. Overlay routing networks, such as RON, have been proposed to monitor path quality and select the best available path via the Internet or a series of RON nodes [2]. However, the required background monitoring is not scalable and therefore limits the approach to communication among a relatively small set of nodes.

This paper re-examines the potential of overlay routing techniques for improving end-to-end Internet path reliability. Our goal is to answer three questions:

1. What do the failure characteristics of wide-area Internet paths imply about the potential reliability benefits of overlay routing techniques?
2. Can this potential be realized with a simple, stateless, and scalable scheme?
3. What benefits would end-users see in practice for a real application, such as Web browsing, when this scheme is used?

To answer the first question, we performed a large-scale measurement study that uses 67 PlanetLab vantage points to probe 3,153 Internet destinations for failures over seven days. Of these destinations, 378 were popular Web servers, 1,139 were broadband hosts, and 1,636 were randomly selected IP addresses. During the course of our 7-day study we observed more failures than the 31 node RON testbed saw in 9 months [7].

Our results show that end-to-end path availability varies substantially across different destination sets. On average, paths to popular Web servers had 99.6% availability, but paths to broadband hosts had only 94.4% availability. The vast majority of paths experienced at least one failure. Unfortunately, many failures are located so close to the destination that no alternative routing or overlay scheme can avoid them: 16% of failures on paths to servers and 60% of failures on paths to broadband hosts occur were last-hop or end-system failures. Effective remedies for these failures are increased end-system reliability and multi-homing.

To answer the second question, we use our measurement results to show that when an alternative path exists, that path can be exploited through extremely simple means. Inspired by the Detour study [21] and RON [2], we explore the use of a technique we call *one-hop source routing*. When a communication failure occurs, the source attempts to reach the destination *indirectly* through a small set of intermediary nodes. We show that a selection policy in which the source node chooses four potential intermediary nodes at random (called *random-4*) can obtain close to the maximum possible benefit. This policy gives well-connected clients and servers the ability to route around failures in the middle of the network *without* the need for complex schemes requiring *a priori* communication or path knowledge.

To answer the third question, we built and evaluated a prototype one-hop source routing implementation called SOSR (for Scalable One-hop Source Routing). SOSR uses the Linux netfilter/iptables facility to implement alternative packet routing for sources and NAT-style forwarding for intermediaries – both at user level. SOSR is straightforward to build and completely transparent to destinations. On a simple workload of Web-browsing requests to popular servers, SOSR with the random-4 policy recovered from 56% of network failures. However, many failures that we saw were application-level failures, such as server timeouts, which are not recoverable through any alternative routing or overlay schemes. Including such application-level failures, SOSR could recover from only 20% of the failures we encountered. The user-level perception of any alternative routing scheme is ultimately limited by the behavior of the servers as well as of the network.

The rest of the paper is organized as follows. We present our measurement study characterizing failures in the next section. Section 3 then shows the potential effectiveness of different practical policies for improving Internet path reliability. Section 4 describes the design, implementation, and evaluation of our prototype one-hop source routing system. We end with a discussion of related work (Section 5) and our conclusions (Section 6).

2 Characterizing Path Failures

This section describes our large-scale measurement study of Internet path failures. Our goals were: (1) to discover the frequency, location, and duration of path failures, and (2) to assess the potential benefits of one-hop source routing in recovering from those failures.

2.1 Trace methodology

From August 20, 2004 to August 27, 2004 we monitored the paths between a set of PlanetLab [18] *vantage points* and sets of destination hosts in the Internet. We periodically sent a probe on each path and listened for a response. If we received a response within a pre-defined time window, we declared the path to be normal. If not, we declared a *loss incident* on that path. Once a loss incident was declared, we began to probe the path more frequently to: (1) distinguish between a "true" path failure and a short-lived congestion episode and (2) measure the failure duration. We used traceroute to determine where along the path the failure occurred. In parallel, we also sent probes from the vantage point to the destination *indirectly* through a set of intermediary nodes. This allowed us to measure how often indirect paths succeeded when the default path failed.

2.1.1 Probes and traceroutes

Our probes consist of TCP ACK packets sent to a high-numbered port on the destination. Correspondingly, probe responses consist of TCP RST packets sent by the destination. We used TCP ACK packets instead of UDP or ICMP probes for two reasons. First, many routers and firewalls drop UDP or ICMP probes, or treat them with lower priority than TCP packets, which would interfere with our study. Second, we found that TCP ACK probes raise fewer security alarms than other probes. Before we included a candidate destination in our study, we validated that it would successfully respond to a burst of 10 TCP ACK probes. This ensured that destinations were not rate-limiting their responses, avoiding confusion between rate-limiting and true packet loss.

To determine *where* a failure occurred along a path, we used a customized version of traceroute to probe the path during a loss incident. Our version of traceroute uses TTL-limited TCP ACK packets, probing multiple hops along the route in parallel. This returns results much faster than the standard traceroute and permits us to determine the location of even short-lived failures.

2.1.2 Node selection

Initially, we selected 102 geographically distributed PlanetLab nodes as vantage points. Following the experiment, we examined the logs on each node to determine which had crashed or were rebooted during our trace. We then filtered the trace to remove any nodes with a total downtime of more than 24 hours, reducing the set to 67 stable vantage points for our analysis. We similarly selected 66 PlanetLab nodes to use as intermediaries, but only 39 of these survived crash/reboot post-filtering.

Using our vantage points, we monitored paths to three different sets of Internet hosts: popular Web servers, broadband hosts, and randomly selected IP addresses. A full list of IP addresses in each set and additional details describing our selection process are available at `http://www.cs.washington.edu/homes/gummadi/sosr`.

The members of each set were chosen as follows:

- We culled our popular server set from a list of the 2,000 most popular Web sites according to www.ranking.com. Removing hosts that failed the TCP ACK rate-limit test and filtering duplicate IP addresses left us with 692 servers. The path behavior to a popular server is meant to be representative of the experience of a client when contacting a well-provisioned server.

- We selected our broadband hosts from an IP address list discovered through a 2002 crawl of Gnutella [20]. From that set, we removed hosts whose reverse DNS lookup did not match a list of major broadband providers (e.g., `adsl*bellsouth.net`) and again filtered those that failed the rate-limit test. Finally, we selected 2,000 nodes at random from those that survived this filtering. The path behavior to a broadband host is meant to be representative of a peer-to-peer application or voice-over-IP (VoiP).

- The random IP address set consists of 3,000 IP addresses that were randomly generated and that survived the rate-limit test. We use this set only as a basis for comparison.

We partitioned the destination sets across our initial vantage points such that each destination node was probed by only one vantage point. Because some of the vantage points were filtered from the trace due to failure or low availability, some of the destinations were consequently removed as well. Following this filtering, 378 servers, 1,139 broadband hosts, and 1,636 random IP addresses remained in the trace. Note that while we filtered vantage points and intermediaries for availability, we did *not* filter any of the destination sets beyond the initial TCP ACK rate-limit test. As a consequence, some destinations crashed or otherwise shut down during the trace, causing last-hop path failures.

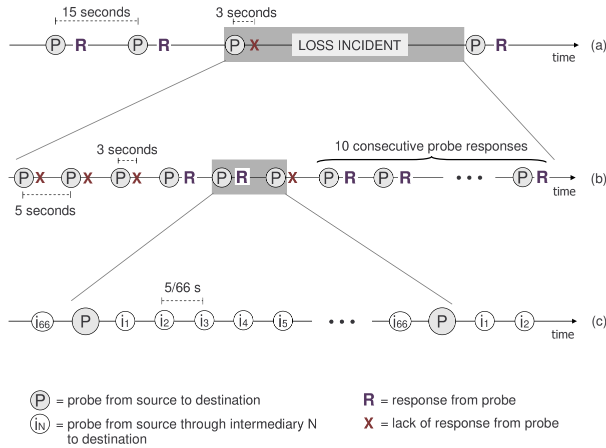


Figure 1: **Probe timing.** (a) The sequence of probes that are sent along each path during the trace. (b) A loss incident begins with a single probe loss, and ends after 10 consecutive successful probe responses. (c) For each of the first 10 probe intervals of a loss incident, we issued indirect probes through each of 66 intermediaries.

2.1.3 Probe timing

During the 7-day trace period, we probed each path every 15 seconds. If the vantage point failed to receive a response within 3 seconds, we declared a loss to have occurred. A single loss transitioned the path into a *loss incident* – an event initiated by a single probe loss and ended by the reception of ten consecutive probe responses (Figure 1a). While a path is in the midst of a loss incident, we probed every 5 seconds (Figure 1b). We also issued a traceroute from the vantage point to the destination at the start of the loss incident.

For each of the first 10 probe intervals during a loss incident, we also attempted to probe the destination *indirectly* through *each* of the 66 PlanetLab intermediaries selected at the beginning of the experiment. Thus, during one of these probe intervals, the vantage point emits a probe to an intermediary every $5/66^{th}$ of a second (Figure 1c). We allow six seconds for a response to flow back from the destination through the intermediary to the vantage point; if no response is received in this time we declare a loss through that intermediary.

2.1.4 Failures vs. loss incidents

In principle, it may seem simple to declare a path failure when some component of a path has malfunctioned and all packets sent on that path are lost. In practice, however, failures are more complex and difficult to define. For example, packet loss may be due to a true long-term failure or a short-term congestion event. In general,

characteristic	servers	broadband	random
paths probed	378	1,139	1,636
vantage points	67	67	67
failure events	1,486	7,560	10,619
failed paths	294	999	1,395
failed links	337	1,052	1,455
classifiable failure events	962	5,723	7,024
last-hop	151 (16%)	3,406 (60%)	2,568 (37%)
non-last-hop	811 (84%)	2,317 (40%)	4,456 (63%)
unclassifiable failure events	524	1,837	3,595

Table 1: **High-level characterization of path failures, observed from 08/20/04 to 08/27/04.** We obtained path information using traceroutes. A failure is identified as a last-hop failure when it is attributable to either the access link connecting the destination to the network or the destination host itself.

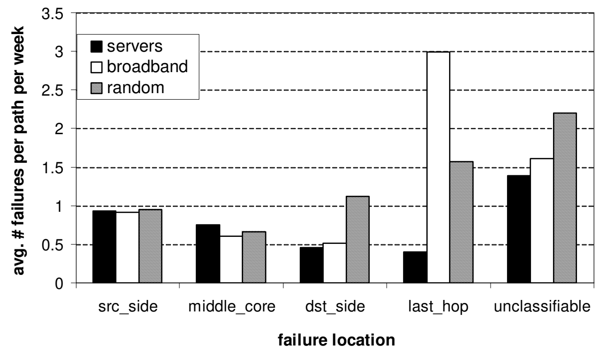


Figure 2: **Location of failures.** Failures are spread throughout the Internet for all three destination sets. Last-hop failures dominate other failures for broadband hosts, but for popular servers, last-hop failures are rare.

any operational definition of failure based on packet loss patterns is arbitrary.

We strove to define failure as a sequence of packet losses that would have a significant or noticeable application impact. We did not want to classify short sequences of packet drops as failures, since standard reliability mechanisms (such as TCP retransmission) can successfully hide these. Accordingly, we elevated a loss incident to a failure if and only if the loss incident began with three consecutive probe losses *and* the initial traceroute failed. We defined the failure to last from the send of the first failed probe until the send of the first of the ten successful probes that terminated the loss incident. For example, the loss incident shown in Figure 1b corresponds to a failure that lasted for 30 seconds.

2.2 Failure characteristics

Table 1 summarizes the high-level characteristics of the failures we observed, broken down by our three destination sets. In the table we show as *classifiable* those failures for which our modified traceroute was able to determine the location of the failure; the remainder we show as *unclassifiable*. The classifiable failures are further broken down into *last-hop failures*, which are failures of either the end-system or last-hop access link (we cannot distinguish the two), and *non-last-hop failures*, which occurred within the network.

For the popular servers, we saw 1,486 failures spread over 294 paths and 337 links along these paths. Of the 962 classifiable failures, 811 (84%) occurred within the network, while 16% were last-hop failures. On average, a path experienced 3.9 failures during the week-long trace, of which 0.4 were last-hop failures, 2.1 were non-last-hop failures, and 1.4 were unclassifiable.

For the broadband hosts we saw 7,560 failures of which 5,723 were classifiable. On average, a broadband path experienced 6.6 failures over the week, nearly double that of the popular server set. Of these 6.6 failures, 3.0 were last-hop, 2.0 non-last-hop, and 1.6 were unclassifiable. Comparing server and broadband paths, we saw approximately the same rate of non-last-hop failures, but broadband paths showed a much higher rate of last-hop failures (0.4 per path per week for servers, and 3.0 per path per week for broadband). Therefore, the network behaved similarly for broadband and popular server hosts, except over the last-hop.

2.2.1 Location of failures

To describe the failure locations in a meaningful way, we divide each path into four parts: *last_hop*, *middle_core*, *src_side*, and *dst_side*. Last_hop are either end-system failures or last-hop access-link failures. Middle_core failures occur in the "core of the Internet," which we define as the *Tier1* domains. These are the few important domains, such as AT&T and Sprint, through which the vast majority of all Internet paths pass. We identify them using the methodology of Subramanian et al. [23]. Src_side and dst_side are therefore the remaining path segments between the core and source, or core and destination, respectively. If traceroute could not classify the failure location, we labeled it "unclassifiable."

Figure 2 shows the distribution of failures across these categories. Failures are spread throughout the Internet, and all three data sets observe approximately equal source-side and core failures. For popular servers, there are relatively few last-hop failures, and in fact the last-hop appears to be more reliable than the rest of the Internet! This is strong evidence that techniques such as one-hop source routing can improve end-to-end availability for server paths, as it targets these non-last-hop failures. For broadband hosts, however, last-hop failures dominate all other failures, and accordingly we should expect less of a benefit from one-hop source routing.

Not surprisingly, the random IP destination set behaves in a manner that is consistent with a blend of

	servers	broadband	random
average path downtime	2,561 secs	33,630 secs	10,904 secs
median path downtime	333 secs	981 secs	518 secs
average failure duration	651 secs	5,066 secs	1,680 secs
last-hop	3,539 secs	8,997 secs	3,228 secs
non-last-hop	339 secs	859 secs	735 secs
unclassifiable	302 secs	3,085 secs	1,744 secs
median failure duration	73 secs	75 secs	72 secs
last-hop	113 secs	100 secs	61 secs
non-last-hop	70 secs	64 secs	66 secs
unclassifiable	70 secs	67 secs	85 secs

Table 2: **Path downtime and failure durations.** This table shows average and median path downtime, as well as average and median failure durations, for our three destination sets. The downtime for a path is the sum of all its failure durations.

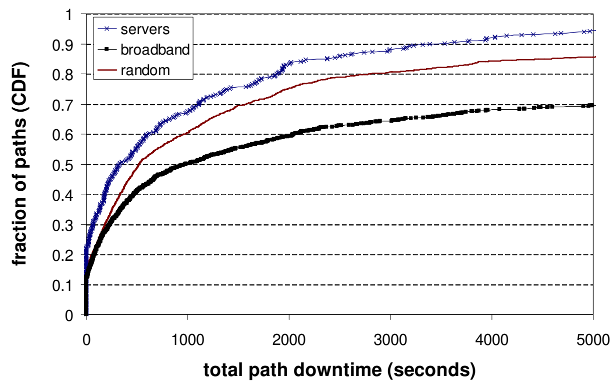


Figure 3: **Availability of paths (CDF).** The cumulative distribution of total downtime experienced by the paths during our trace, for each of our three destination sets.

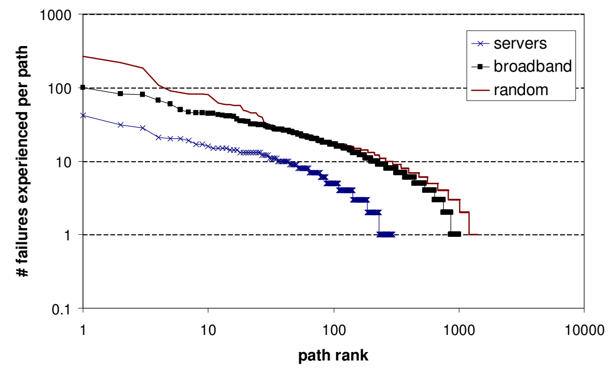


Figure 4: **Frequency of path failures.** Paths are ordered by the number of failures experienced. Most paths experience few failures, but a small number of paths experience many failures. Note that both axes are in log-scale.

server-like and broadband-like hosts. Somewhat surprisingly, however, the random IP set sees a greater rate of destination-side failures than both servers and broadband hosts. We do not yet have an explanation for this.

2.2.2 Duration of failures

In Table 2, we show high-level statistics that characterize failure duration and path availability in our trace. The average path to a server is down for 2,561 seconds during our week long trace, which translates into an average availability of 99.6%. In comparison, the average path to a broadband host is down for 33,630 seconds during trace, leading to an average availability of 94.4%. Paths to broadband hosts are an order of magnitude less available than paths to server hosts. This is unsurprising, of course, since broadband hosts are less well maintained, tend to be powered off, and likely have worse quality last-hop network connections.

The median path availability is significantly better than the average path availability, suggesting that the distribution of path availabilities is non-uniform. Figure 3 confirms this: for all three destination sets, more than half of the paths experienced less than 15 minutes of downtime over the week. As well as being generally less available than server paths, a larger fraction broadband paths suffer from high unavailability: more than 30% of broadband paths are down for more than an hour, and 13% are down for more than a day (not shown in graph).

Table 2 also shows the average and median failure durations. On paths to servers, the average failure lasted for just under 11 minutes; in comparison, on paths to broadband hosts, the average failure lasted for 84 minutes. For both destination sets, last-hop failures lasted approximately an order of magnitude longer than non-last-hop failures. Unfortunately, this reduces the potential effectiveness of one-hop source routing. Last-hop failures can last for a long time, and they are also hard to route around. Like path availability, failure duration has a non-uniform distribution – median failure durations are significantly lower than average failure durations.

2.2.3 Frequency of failures

We can compute the number of failure-free paths by subtracting the number of failed paths from the number of paths probed (both shown in Table 1). This reveals that only 22% of server paths, 12% of broadband paths, and 15% of random paths were failure-free: most paths in each destination set experienced at least one failure.

Figure 4 plots the number of failures each path experienced, using a log-scale on both axes. Each destination set is sorted in rank order from most-failing to least-failing path. This graph demonstrates two points: (1) a small number paths experience a very large number of failures, and (2) most paths experienced a small but nonzero number of failures. Additional analysis (not shown) also demonstrates that for broadband hosts, the failure-prone paths tend to fail on the last-hop, while for servers, the failure-prone paths tend to fail uniformly across the

recoverable failures	servers	broadband	random
src_side	54% (189 of 353)	55% (577 of 1042)	54% (838 of 1548)
middle_code	92% (262 of 284)	90% (616 of 686)	94% (1017 of 1078)
dst_side	79% (138 of 174)	66% (391 of 589)	65% (1202 of 1830)
last_hop	41% (62 of 151)	12% (406 of 3406)	24% (606 of 2568)
unclassifiable	63% (332 of 524)	54% (983 of 1837)	58% (2096 of 3595)
all	**66% (983 of 1486)**	**39% (2973 of 7560)**	**54% (5759 of 10619)**

Table 3: **Potential effectiveness of one-hop source routing.** Source routing can help recover from 66% of all failures on paths to servers, but fewer on paths to broadband hosts. Last-hop failures tend to confound recovery, while core failures are more recoverable.

Internet, favoring neither the source-side, nor the core, nor the destination-side.

2.3 The potential of one-hop source routing

As previously described, during a loss incident we probed the destination indirectly through each of 39 intermediaries. If any of these indirect probes were successful, we considered the path to be recoverable using one-hop source routing. If not, we considered it to be unrecoverable. Note that this definition of recoverable provides an upper bound, since in practice an implementation is not likely to try such a large number of intermediaries when attempting to route around a failure.

The results of this experiment, shown in Table 3, indicate that 66% of all failures to servers are potentially recoverable through at least one intermediary. A smaller fraction (39%) of broadband failures are potentially recoverable. For all destination sets, one-hop source routing is very effective for failures in the Internet core, but it is less effective for source-side or destination-side failures. Somewhat surprisingly, some last-hop failures are recoverable. In part, this is due to multi-homing: i.e., there may be a last-hop failure on the default path to a destination, but a *different* last-hop link may be accessible on a different path through a destination. However, this is also due in part to our failure definition. If a last-hop link is not dead but merely "sputtering," sometimes probes along the default path will fail while an intermediary will be more "lucky" and succeed.

2.4 Summary

Our study examined failures of Internet paths from 67 vantage points to over 3,000 widely dispersed Internet destinations, including popular servers, broadband hosts, and randomly selected IP addresses. Overall, we found that most Internet paths worked well: most paths only experienced a handful of failures, and most paths experienced less than 15 minutes of downtime over our week-long trace. But failures do occur, and when they do, they were widely distributed across paths and portions of the network. However, broadband hosts tend to experience significantly more last-hop failures than servers, and last-hop failures tend to last long.

These failure characteristics have mixed implications for the potential effectiveness of one-hop source routing. Since server path failures are rarely on the last hop, there should be plenty of opportunity to route around them. Indeed, our initial results suggest that one-hop source routing should be able to recover from 66% of server path failures. In contrast, since broadband path failures are often on the last hop, there is less opportunity for alternative routing. Our results show than one-hop source routing will work less than 39% of the time in this case.

In the next section of the paper, we will examine one-hop source routing in greater detail, focusing initially on its potential for improving server path availability. Our goal in the next section is to use our trace to hone in on an effective, but practical, one-hop source routing policy. By effective, we mean that it successfully routes around recoverable failures. By practical, we mean that it succeeds quickly and with little overhead.

3 One-Hop Source Routing

We have seen that 66% of all popular server path failures and 39% of all broadband host path failures are potentially recoverable through at least one of the 39 preselected intermediary nodes. This section investigates an obvious implication of this observation, namely, that *one-hop source routing* is a potentially viable technique for recovering from Internet path failures.

One-hop source routing is conceptually straightforward, as shown in Figure 5. After a node detects a path failure, it selects one or more intermediaries and attempts to reroute its packets through them. If the resulting indirect path is sufficiently disjoint from the default route, the packets will flow around the faulty components and successfully arrive at the destination. Assuming that the reverse path through the intermediary also avoids the fault, end-to-end communication is restored.

This approach raises several questions. Given a set of potential intermediaries for a failed path, how many of them on average will succeed at contacting the destination? What policy should the source node use to select among the set of potential intermediaries? To what extent does the effectiveness of one-hop source routing depend on the location of the failure along the path? Does *a priori* knowledge of Internet topology or the ability to maintain state about previous failures increase the effectiveness of a policy? When should recovery be initiated

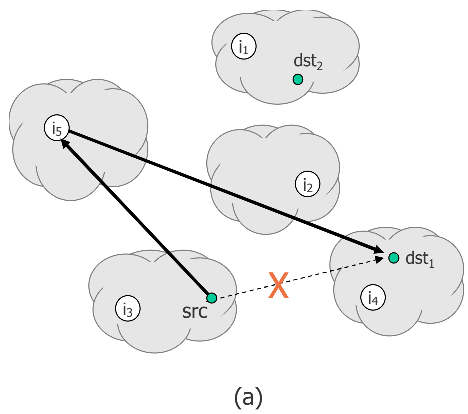


Figure 5: **One-hop source routing.** (a) The source (src) experiences a path failure to destination dst_1, but it successfully routes through intermediary i_5. (b) The source experiences a path failure to destination dst_2. It uses a more aggressive recovery policy by simultaneously routing to intermediaries i_2, i_3, i_4, and i_5. The path to intermediary i_2 experiences a path failure of its own, as does the path from intermediary i_3 to the destination. Fortunately, the source is able reach dst_2 through both i_4 and i_5.

and when should recovery attempts be abandoned? The remainder of this section answers these questions.

3.1 Methodology

To answer these questions we rely on the data described in Section 2. As previously noted, following each failure we sent probe messages from the source to 39 PlanetLab intermediaries. The intermediaries then probed the destination and returned the results. If the source heard back from an intermediary *before* it heard back directly from the (recovered) destination, then we considered that intermediary to be successful. Thus, for each default-path failure, we were able to determine *how many* of the 39 PlanetLab intermediaries *could have been used* to route around it.

From this data we can analyze the effectiveness of policies that route through specific subsets of the intermediaries. For example, one policy might route through a single, randomly chosen intermediary; another policy might route through two preselected intermediaries in parallel, and so on. We can therefore compare various policies by simulating their effect using the data from our intermediate-node measurements.

3.2 What fraction of intermediaries help?

How many of the intermediaries succeed in routing around a particular failure depends on a number of factors, including the positions of the source, the destination, and the intermediaries in the network. For example, some intermediaries may not divert the packet flow sufficiently, either failing to pull packets from the default path before the fault or failing to return them to the default path after the fault. This can be seen in Figure 6a, where the route from src to dst fails due to the failure

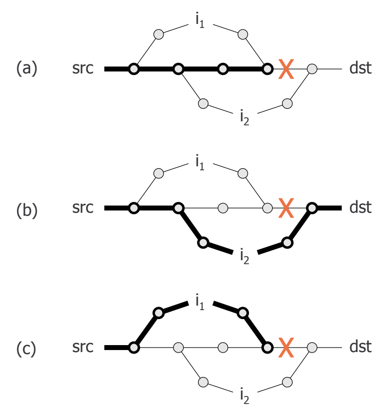
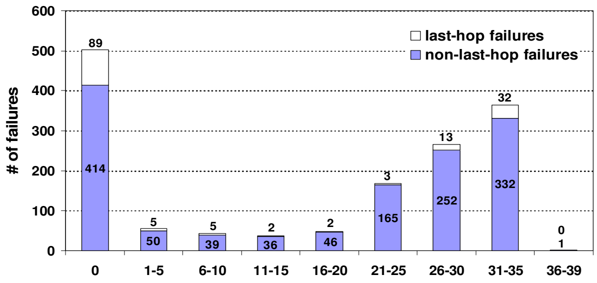


Figure 6: **Disjoint paths.** (a) The default path to the destination fails due to a faulty link. (b) Routing through intermediary i_2 would succeed, since the diverted path is disjoint from the faulty link. (c) Routing through intermediary i_1 would fail, since the diverted path rejoins the default path before the faulty link.

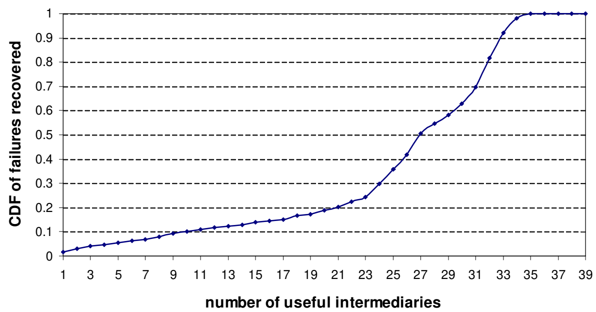
marked "X." An attempt to re-route through intermediary i_2 would succeed (Figure 6b). However, routing through i_1 would fail (Figure 6c), because i_1's path to dst joins src's path to dst *before* the failure.

As described above, for each detected failure we counted the number of "useful intermediaries" through which the source node could recover. Note that we continue attempting to recover until either an intermediary succeeds or the default path self-repairs, up to a maximum of 10 probe intervals. If the default path self-repairs before any intermediary succeeds, we do not classify the event as a recovered failure.

Figure 7(a) shows the results for popular servers, grouped by the number of useful intermediaries on the



(a)



(b)

Figure 7: **The number of useful intermediaries.** For each failure, we measured the number of useful intermediaries in our candidate set of 39. (a) This histogram shows the aggregated results for popular servers only. For example, we observed 168 failures (165 non-last-hop and 3 last-hop) for which there were exactly 21-25 useful intermediaries. (b) A CDF of the same data for the recoverable failures only.

x-axis. Out of the 1486 failures, 503 (34%) could not recover through any intermediary, as shown by the leftmost bar. Last-hop failures accounted for 89 of those unrecoverable failures.

Figure 7(b) presents a CDF of this data for the remaining 983 failures (66%) that *were* recoverable. The figure shows that 798 (81%) of these failures could be recovered through *at least* 21 of the 39 intermediaries. It's clear, then, that a significant fraction of failures are recoverable through a large number of intermediaries. However, there are also failures for which only a small number of intermediaries are useful. For example, 55 (5.6%) of the 983 recoverable failures could be recovered through only 1-5 nodes. Thus, some recoverable failures require a careful choice of intermediary.

None of the failures were recoverable through more than 36 of the 39 intermediaries. Investigating further, we found that four PlanetLab intermediaries were subject to a routing policy that prevented them from communicating with the vast majority of our destinations. If we exclude these nodes from consideration, many failures would be recoverable through all 35 of the remaining intermediaries. However, in many cases, there were still a few intermediaries that should be avoided.

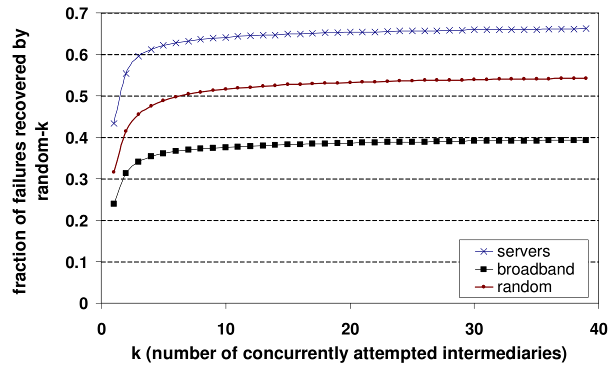


Figure 8: **The effectiveness of random-k.** This graph shows the effectiveness of random-k at recovering from failures as a function of k (the number of randomly selected intermediaries the source tries concurrently).

3.3 Is random-k an effective policy?

The results in Figure 7 suggest that a very simple strategy for selecting intermediaries may work well. Similar in spirit to randomized load-balancing [15, 6], a source should be able to avoid failures by randomly picking k intermediaries through which to attempt recovery. The source could send packets through all k intermediaries in parallel and then route through the intermediary whose response packet is first returned.

To evaluate this strategy, we examine a policy in which the random selection is done once for each failure instance. When a failure is detected, the source selects a set of k random intermediaries. During the failure, if none of the k intermediaries succeed on the first attempt, the source continues to retry those same intermediaries. At the next failure, the source selects a new set of k random intermediaries. We call this policy *random-k*.

Since many intermediaries can be used to avoid most recoverable faults, even a single random selection (random-1) should frequently succeed. By selecting more than one random intermediary, the source ensures that a single unlucky selection is not fatal. However, as there are some failures for which only a few specific intermediaries are helpful, picking a small number of random intermediaries will not always work.

Figure 8 shows the effectiveness of random-k as a function of k. For popular servers, random-1 can route around 43% of all failures we observed. By definition, random-39 can route around all recoverable failures (66% of all failures for popular servers). The "knee in the curve" is approximately $k = 4$: random-4 can route around 61% of all failures (92% of all recoverable failures) for popular servers. From this, we conclude that random-4 makes a reasonable tradeoff between effort (the number of concurrent intermediaries invoked per re-

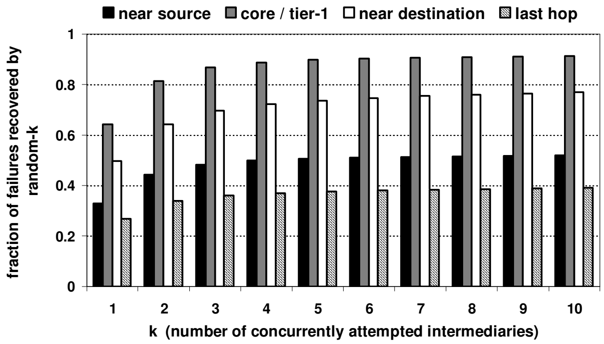


Figure 9: **Failures random-k can handle.** This graph breaks Figure 8 data down according to the classes of failures from which random-k recovers, for servers. Random-k can recover from most failures in the core, and near the destination, but it is less capable at handle failures near the source or on the last hop.

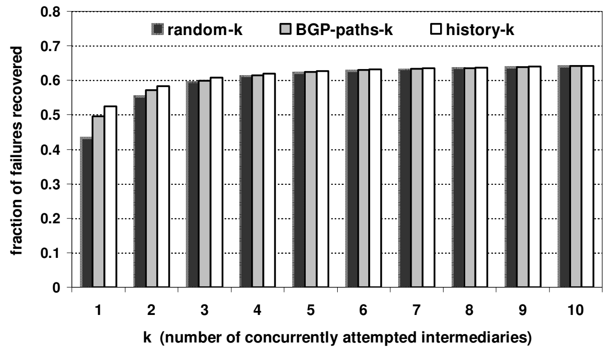


Figure 10: **Effectiveness of alternative policies.** This graph shows the effectiveness of our alternative policies as a function of k. We include random-k for comparison.

covery attempt) and the probability of success.

3.4 Which failures can random-k handle?

As we showed in Section 2, the location of a failure has a significant impact on the likelihood that one-hop source routing can recover. For example, last-hop failures are much harder to recover from than core failures. To understand the impact of failure location on random-k's ability to recover, we classified recovery attempts according to the failure location. Figure 9 shows the same data as Figure 8 broken down according to this classification, but for popular servers only.

Random-k recovers poorly from near-source and last-hop failures, as shown in the figure. For example, random-4 recovers from only 37% of last-hop failures and 50% of near-source. However, random-4 is *very successful* at coping with the other failure locations, recovering from 89% of middle_core and 72% of near-destination failures. Intuitively, the Internet core has significant path diversity, therefore a failure in the core is likely to leave alternative paths between many intermediaries and the source and destination. However, the closer the failure is to the source or the destination, the more intermediaries it will render ineffective.

3.5 Are there better policies than random-k?

Figure 8 shows that random-k is a very effective policy: there is little room to improve above random-k before "hitting the ceiling" of recoverable failures. But can we be smarter? This subsection explores two alternative policies that use additional information or state to further improve on random-k. These policies might not be practical to implement, as they require significant amounts of prior knowledge of Internet topology or state. Nonetheless, an analysis of these policies offers additional insight into one-hop source routing. Like random-k, each of the alternative policies selects k intermediaries through which to route concurrently for each recovery attempt. The two additional policies we consider are:

1. **History-k.** In this policy, we assume that the source node remembers the intermediary that it most recently used to recover from a path failure for each destination. When the source experiences a path failure, it selects $k-1$ intermediaries at random, but it chooses this recently successful intermediary as the k^{th} intermediary. If the source has never experienced a failure for the destination, this policy reverts to random-k. The rationale for this policy is that an intermediary that previously provided a sufficiently disjoint path to a destination is likely to do so again in the future.
2. **BGP-paths-k.** In this policy, we assume that the source is able to discover the path of autonomous systems (ASs) between it and the destination, and between all intermediaries and the destination, as seen by BGP. For each intermediary, the source calculates how many ASs its path has in common with the intermediary's path. When the source experiences a path failure to the destination, it orders the intermediaries by their number of common ASs and selects the k intermediaries with the smallest number in common. The rationale for this policy is that the source wants to use the intermediary with the "most disjoint" path to the destination in an attempt to avoid the failure.

Figure 10 shows how effective these policies were in recovering from failures. While there is some measurable difference between the policies for low values of k, this difference diminishes quickly as k increases. There is some benefit to using a more clever policy to pick the "right" intermediary; however, slightly increasing the number of intermediaries is more effective than using a

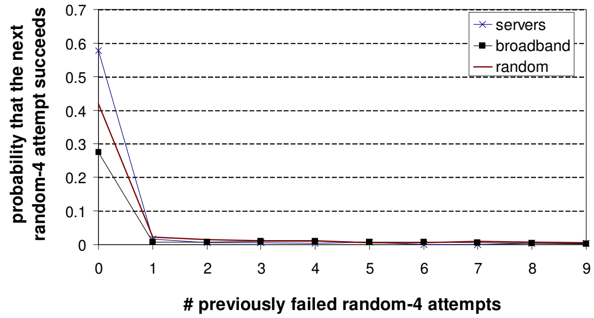


Figure 11: **The diminishing returns of repeated recovery attempts.** The probability that the next random-4 attempt will succeed, as a function of the number of previously failed attempts. After a single attempt, the probability of future success plummets.

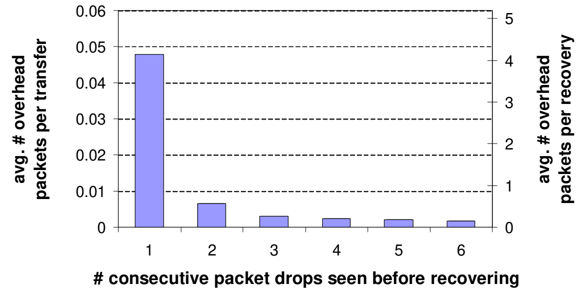


Figure 12: **The cost of being eager.** Average number of overhead packets per connection and per recovery invocation, as a function of the number of consecutive packet drops seen before random-4 recovery is invoked. Because failures happen rarely, recovery is inexpensive.

smarter policy. For example, though BGP-paths-1 beats random-1, it does not beat random-4. Furthermore, unlike BGP-paths, random-4 requires no prior knowledge and no overhead for acquiring that knowledge.

These more sophisticated policies do have some positive effect on the ability to recover from path failures. They essentially attempt to bias their intermediary selection towards "better" intermediaries, and this biasing has some value. However, the goal of path failure recoverability is to avoid bad choices, rather than finding the best choice. Even with small k, random-k is extremely unlikely to select only bad choices, which is why it is competitive with these other strategies.

3.6 How persistent should random-4 be?

So far, we allow a source node recovering from a path failure to continue issuing random-4 attempts every 5 seconds until: (1) one of the four randomly selected intermediaries succeeds, (2) the path self-repairs, or (3) ten attempts have failed. We now consider the question of whether the source node should give up earlier, after fewer attempts. To answer this, we calculated the probability that the next random-4 attempt would succeed but the default path remains down, as a function of the number of previously failed random-4 attempts.

Figure 11 shows the results. Immediately after noticing the failure, random-4 for popular servers has a 58% chance of recovering before the path self-repairs; for broadband hosts, this number is 28%. However, after a single failed random-4 attempt, the chance that the next attempt succeeds before the path self-repairs plummets to 1.6% for servers and 0.8% for broadband hosts. There is little reason to try many successive random-4 attempts; the vast majority of the time, the default path will heal before a future random-4 attempt succeeds.

In a sense, random-4 is an excellent failure detector: if a random-4 attempt fails, it is extremely likely that the destination truly is unreachable. To be conservative, though, we allow four random-4 attempts before giving up on intermediary-based recovery.

3.7 How eager should random-4 be?

Our methodology defines a failure to have occurred after three consecutive packet drops and a failed traceroute. There is nothing preventing a source from invoking random-4 before making this determination, however. For example, a source may choose to invoke random-4 after seeing only a single packet drop. However, this approach might potentially confuse short-term congestion for failure and needlessly introduce recovery overhead.

To investigate this apparent tradeoff between recovery time and overhead, we used our measurement data to calculate the overhead of recovery relative to hypothetical background HTTP traffic. To do this, we consider each 15-second probe attempt to be a hypothetical HTTP request. We estimate that the transfer of a 10KB Web page, including TCP establishment and teardown, would require 23 IP packets. Using this estimate, we can calculate the amount of packet overhead due to random-4 recovery attempts.

Figure 12 shows this overhead as a function of how eagerly recovery is invoked. The left-hand y-axis shows the average number of additional packets sent per HTTP transfer, and the right-hand y-axis shows the average number of additional packets sent per transfer for which recovery was invoked. As our previous results suggested, we abandon after four repeated random-4 attempts.

Failures occur rarely and there is no overhead to pay when the default path succeeds. Additionally, when failures do occur, random-4 usually succeeds after a single attempt. For these two reasons, there is very little overhead associated with the average connection: only 0.048 additional packets per connection on average (on top of the estimated 23 IP packets), assuming recovery begins immediately after a lost packet. Even when failures occur, there are only 4.1 additional packets on average.

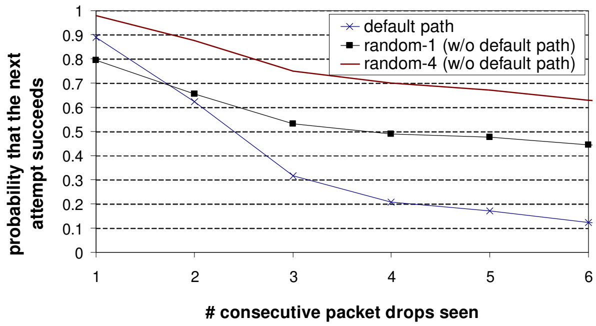


Figure 13: **The benefit of being eager.** This graph compares the likelihood that the default-path, a single-intermediary, and four intermediaries will succeed as a function of the number of consecutive packet drops observed. After just two observed packet drops, random-4 has a significantly higher probability of success than the default path.

There is therefore little cost to being eager.

Figure 13 demonstrates why it is useful to recover eagerly. We compare the probability that on the next attempt (1) the default-path will succeed, (2) a single randomly chosen intermediary will succeed, and (3) four randomly chosen intermediaries will succeed, as a function of the number of previously observed consecutive packet drops along the default path. After a single packet drop, all three strategies have an equally high probability of succeeding on the next attempt. But, with additional packet drops, the probability that the default path succeeds quickly decays, while random-1 and random-4 are more likely to work.

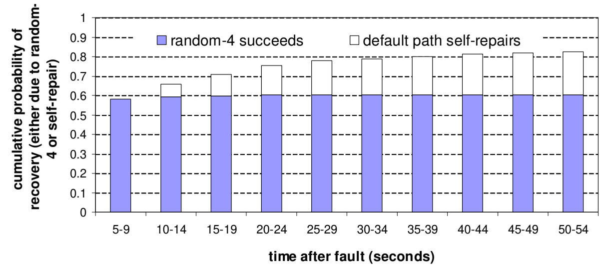
Note that random-1 has a slightly lower probability of succeeding than the default path after only one packet drop. For random-1, both the source to intermediary and the intermediary to destination paths must work, while for default-path, only the source to destination path must work.

There is a benefit to recovering early, and as we previously showed, there is very little cost. Accordingly, we believe random-4 should be invoked after having observed just a single packet drop.

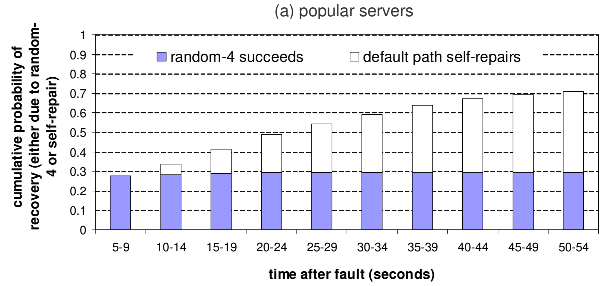
3.8 Putting it all together

Our results suggest that an effective one-hop source routing policy is to begin recovery after a single packet loss, to attempt to route through both the default path and four randomly selected intermediaries, and to abandon recovery after four attempts to each of the randomly selected intermediaries fail, waiting instead for the default path to recover itself. For the remainder of this paper, we will refer to this set of policy decisions as "random-4."

We now ask what the user experience will be when using random-4. To answer this, we measured how long the user must wait after experiencing a path failure for end-to-end connectivity to be re-established, due either to the success of random-4 or path self-repair.



(a) popular servers



(b) broadband hosts

Figure 14: **Recovery latency.** This graph plots the cumulative probability that a failed path has recovered – due either to random-4 succeeding or self-repair of the default path – as a function of time, for (a) popular servers, (b) broadband hosts.

Figure 14 shows the results. For popular servers, we see that 58.1% of failures recover after a single attempt (five seconds). After four attempts (20 seconds), 75.3% of path failures recovered. Random-4 recovered from 60.2% of the failures by that point, while the path self-repaired from 15.1% of the failures. Since we abandon random-4 after four failed attempts, the fraction of paths that recover due to random-4 peaks at 60.2%. For broadband hosts, we see a similar pattern, except that random-4 is much less successful at recovering from failures: after four attempts, random-4 recovery has peaked at 29.4%.

Overall, we see that the major benefit of random-4 one-hop source routing is that it quickly finds alternative paths for recoverable failures. However, many failures are not recoverable with one-hop source routing. Bounding repeated random-4 to four attempts restricts the effort expended, instead allowing the path to self-repair.

3.9 Summary

This section examined specific policies for exploiting the potential benefit of one-hop source routing. Our results show that a simple, stateless policy called *random-4* comes close to obtaining the maximum gain possible. For example, from our trace data, random-4 found a successful intermediary for 60% of popular server failures (out of the 66% achievable shown previously in Section 2). Furthermore, random-4 is scalable and requires no overhead messages. In comparison, the alternative knowledge-based policies we examined have higher cost and only limited benefit, relative to random-4.

The crucial question, then, is whether one-hop source routing can be implemented practically and can work effectively in a real system. We attempt to answer this question in the next section.

4 An Implementation Study

The goals of this section are twofold: (1) to discuss the pragmatic issues of integrating one-hop source routing into existing application and OS environments, and (2) to evaluate how well it works in practice. To this end, we developed a one-hop source routing prototype, which we call SOSR (for **s**calable **o**ne-hop **s**ource **r**outing), and evaluate the prototype experimentally in the context of a Web-browsing application for popular servers.

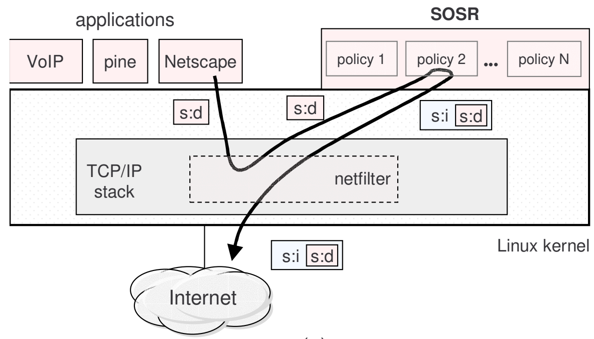
4.1 Prototype Implementation

The high-level SOSR architecture consists of two major parts: one for the *source-node* and one for the *intermediary-node*. The source-node component retries failed communications from applications through one or more intermediaries. For scalability, all decision making rests with the source. The intermediate-node component forwards the messages it receives to the destination, acting as a proxy for the source. There is no destination-node component; SOSR is transparent to the destination.

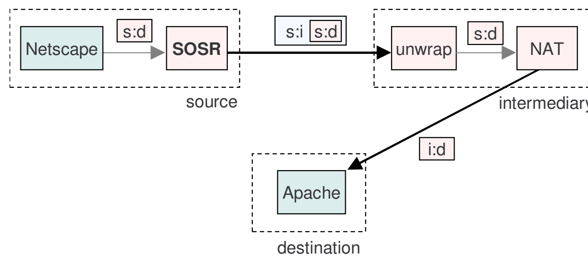
Our source-node component, shown in Figure 15a, is implemented on top of the Linux *netfilter* framework [16]. Netfilter/iptables is a standard feature of the Linux kernel that simplifies construction of tools such as NATs, firewalls, and our SOSR system. It consists of an in-kernel module that forwards packets to a user-mode module for analysis. After receiving and examining a packet, the user-mode module can direct netfilter on how to handle it, acting in essence as an intelligent IP-level router.

We use the iptables rule-handling framework to inform netfilter about which packets to redirect to the user-level SOSR code and which to pass through. Our rules cause specific TCP flows (messages in both directions) to be redirected, based on port numbers or IP addresses. As well, they associate each flow with a SOSR policy module to handle its messages. Each policy module consists of failure detection code and intermediary selection code that decides when and where to redirect a packet.

The SOSR intermediary node acts as a NAT proxy [9], forwarding packets received from a source to the correct destination, and transparently forwarding packets from the destination back to the correct source. Figure 15b shows the encapsulation of indirected messages flowing through the intermediary. The source node encapsulates



(a)



(b)

Figure 15: **The SOSR architecture.** (a) The source node uses netfilter to redirect packets from the Linux kernel to a user-level SOSR policy module, which encapsulates the application packet and tunnels it to a chosen intermediary. Note that the application is unmodified. (b) The intermediary node decapsulates the packet, passes it through a user-level NAT daemon, and transmits it to the (unmodified) destination node.

a TCP segment and information identifying the destination, and tunnels it to the intermediary through UDP. On receiving the UDP packet, the intermediary unwraps the TCP segment, passes it through its NAT component, and sends it to the destination using raw sockets. The NAT component of the intermediary maintains a hash table associating the sending node, the destination node, and the sending port it used to forward the segment. When a packet arrives back from the destination, the intermediary finds the associated (original) source address in the table, encapsulates the segment in a UDP packet along with the destination address, and returns it to the original source.

4.2 Evaluation Methodology

We evaluate our SOSR prototype by measuring its effectiveness in handling end-to-end Web-browsing failures for a set of popular Web servers. These failures include path failures, end-host failures, and application-level failures – anything that disrupts the browser from successfully completing an HTTP server transaction. The destination set consisted of 982 of the popular Web servers we considered in our measurement trace.

For our experiments, we used the same 39 SOSR intermediaries as in our earlier trace-based experiment. We then used three client machines at the University of Washington to repeatedly fetch Web pages from the destination set. Each machine ran a different error-handling protocol. The first simply ran the *wget* command-line Web browser on an unmodified Linux kernel. The second ran the same *wget* browser, but with the SOSR implementation in place, implementing the random-4 policy described in Section 3. The third ran *wget* on a modified, aggressive TCP stack; in place of standard TCP exponential back-off, we sent five duplicate packets every three seconds on a packet loss. This provides a balanced comparison to SOSR, in that it sends packets in the same number and frequency as random-4, but without the path diversity provided by the intermediaries.

Each client machine fetched a Web page once per second, rotating through the 982 Web servers a total of 279 times, ultimately issuing 273,978 requests. The machines fetched the same Web page at the same time, so that a path failure that affected one would hopefully affect the others as well. We ran our experiment for slightly more than 72 hours.

4.2.1 Failure classification

We classified failures as either *network-level failures* or *application-level failures*. If *wget* could not establish a TCP connection to the remote server, we classified the request as a network-level failure. However, if the destination returned a TCP RST packet to the source to refuse the connection, we classified the request as a "TCP refused" application-level failure, since the network path was clearly working.

If *wget* successfully established a TCP connection, but could not successfully complete an HTTP transaction within 15 minutes, we classified the request as an application-level failure. We further sub-classify application-level failures according to whether the HTTP transaction timed out ("HTTP timeout") or was dropped by the server before completion ("HTTP refused"). Though we do not consider them to be failures, we also noted HTTP response that contained an HTTP error code, such as "HTTP/1.1 500 server error."

4.3 Results

Table 4 summarizes our results. Overall, we observed few failures in our 72-hour experiment. The default *wget* client saw 481 failures in 273,978 requests, a failure rate of only 0.18%. We classified 69% as network-level failures and 31% as application-level failures. Thus, the network was responsible for approximately twice as many failures as the Web servers. However, indirect routing cannot recover from the Web server failures.

	requests	failures	network level failures	application level failures			HTTP error codes
				TCP refused	HTTP refused	HTTP timeout	
wget	273,978	481 (0.18%)	328 (0.12%)	40 (0.01%)	78 (0.03%)	35 (0.01%)	44 (0.02%)
wget SOSR	273,978	383 (0.14%)	145 (0.05%)	41 (0.01%)	101 (0.04%)	96 (0.04%)	37 (0.01%)
wget aggTCP	273,978	486 (0.18%)	246 (0.09%)	43 (0.02%)	109 (0.04%)	88 (0.03%)	38 (0.01%)

Table 4: **Failures observed.** Summary of our SOSR prototype evaluation results, showing the number of network-level and application-level failures each of our three test clients observed.

In comparison, the *wget-SOSR* client experienced 20% fewer failures than the default wget client: 383 compared to 481. Of these, 145 (38%) were network-level failures and 238 (62%) were application-level failures. *wget-SOSR* recovered from many of the network-level failures that the default *wget* experienced. The network-level recovery rate for *wget-SOSR* was 56%, slightly below the 66% we measured in Section 3. However, since application-level errors are unrecoverable, *wget-SOSR*'s overall recovery rate was just 20%.

The *wget-aggressiveTCP* client experienced a marginally higher number of failures than the default *wget*: i.e., an aggressive TCP stack did not reduce the overall failure rate. However, it did reduce the number of network-level failures by 25%. The aggressive retransmissions were able to deal with some of the failures, but not as many as *wget-SOSR*. This confirms that SOSR derives its benefit in part because of path diversity, and in part because of its more aggressive retransmission.

Interestingly, both *wget-SOSR* and *wget-aggressiveTCP* observed a higher application-level failure rate than the default *wget*, receiving a significant increase in HTTP refused and timeout responses. While we cannot confirm the reason, we suspect that this is a result of the stress that the additional request traffic of these protocols causes on already overloaded servers. These additional application-level failures in balance reduced (for *wget-SOSR*) or canceled (for *wget-aggressiveTCP*) the benefits of the lower network-level failure rate.

4.4 Summary

This section presented a Linux-based implementation of one-hop source routing, called SOSR. SOSR builds on existing Linux infrastructure (netfilter/iptables) to route failures through a user-mode policy module on the source and a user-mode NAT proxy on intermediaries.

Our SOSR implementation was able to reduce recoverable (network-level) failures by 56% – close to what was predicted by our trace. However, with the Web-browsing application, we saw a new class of failures caused by the Web servers themselves rather than the network; these application-level failures cannot be recovered using indirect network routing. Including these non-recoverable failures, our SOSR implementation was able to reduce the end-to-end failure rate experienced by Web clients by only 20%.

Given the extremely low failure rate that non-SOSR Web clients experience today, we do not believe that SOSR would lead to any noticeable improvement in a person's Web browsing experience. However, this does not imply that one-hop source routing is without value. SOSR is very successful at routing around non-last-hop network failures, and moreover, it has very little overhead. An application that requires better path availability than the Internet currently provides can achieve it using one-hop source routing, assuming that the paths it communicates over have relatively few last-hop failures. Finally, it is likely that SOSR achieves close to the maximum achievable benefit of alternative routing; overlay schemes that attempt to improve reliability are unlikely to better SOSR's recovery rate, and have significant scalability problems due to high message overhead as well.

5 Related Work

Internet reliability has been studied for at least a decade. Paxson's study of end-to-end paths found the likelihood of encountering a routing pathology to be 1.5% in 1994 and 3.3% in 1995 [17]. Zhang concluded that Internet routing had not improved five years later [26]. Chandra observed an average wide-area failure rate of 1.5%, close to "two 9s" of availability [5]. Labovitz's study of routing found path availability to vary widely between "one 9" and "four 9s" [13]. These studies all demonstrate that Internet reliability falls short of that measured for the telephone network [11].

Feamster characterized path failures between overlay nodes. He found wide variation in the quality of paths, failures in all regions of the network, and many short failures [7]. Our work offers a more comprehensive study of network failures that measures paths to a much larger set of Internet-wide destinations. Our results are largely consistent with these earlier findings.

Server availability may be improved using content distribution networks (CDNs) [10] and clusters [8, 4]. This is common for high-end Web sites. However, unlike our technique, it is applicable only to particular services, such as the Web.

Akella uses measurements to confirm that multi-homing has the potential to improve reliability as well as performance [1]. However, a strategy is still needed to select which path to use to obtain these benefits. Our technique is one such strategy; it will take advantage of multi-homing when it exists. There are also commercial products that operate at the BGP routing level to select paths [19, 22]. The advantage of operating at the packet level, as we do, is more rapid response to failures. Conversely, it is known that BGP dynamics can result in a relatively long fail-over period [12] and that BGP misconfigurations are common [14].

Our work is most closely related to overlay routing systems that attempt to improve reliability and performance. The Detour study suggested that this could be accomplished by routing via intermediate end-systems [21]. RON demonstrated this to be the case in a small-scale overlay [2, 3]. Our work differs in two respects. First, we target general communication patterns rather than an overlay. This precludes background path monitoring. Second, and more fundamentally, we show that background monitoring is not necessary to achieve reliability gains. This eliminates overhead in the common no failure case. Our finding is consistent with the study by Teixeira [24] that observed a high-level of path diversity that is not exploited by routing protocols. The NATRON system [25] uses similar tunneling and NAT mechanisms as SOSR to extend the reach of a RON overlay to external, RON-oblivious hosts.

6 Conclusions

This paper proposed a simple and effective approach to recovering from Internet path failures. Our approach, called *one-hop source routing*, attempts to recover from path failures by routing indirectly through a small set of randomly chosen intermediaries. In comparison to related overlay-based solutions, one-hop source routing performs no background path monitoring, thereby avoiding scaling limits as well as overhead in the common case of no failure.

The ability to recover from failures only matters if failures occur in practice. We conducted a broad measurement study of Internet path failures by monitoring 3,153 randomly selected Internet destinations from 67 PlanetLab vantage points over a seven day period. We observed that paths have relatively high average availability (99.6% to popular servers, but only 94.4% for broadband), and that most monitored paths experienced at least one failure. However, 16% of failures on paths to servers and 60% of failures on paths to broadband hosts were located on the last-hop or end-host. It is impossible to route around such last-hop failures. Overall, our measurement study demonstrated that a simple one-hop source routing technique called "random-4" could recover from 61% of path failures to popular servers, but only 35% of path failures to broadband hosts.

We implemented and deployed a prototype one-hop source routing infrastructure on PlanetLab. Over a 48 hour period, we repeatedly accessed 982 popular Web servers and used one-hop source routing to attempt to route around failures that we observed. Our prototype was able to recover from 56% of network failures, but we also observed a large number of server failures that cannot be addressed through alternative routing techniques. Including such application-level failures, our prototype was able to recover from 20% of failures encountered.

In summary, one-hop source routing is easy to implement, adds negligible overhead, and achieves close to the maximum benefit available to any alternative routing scheme, *without* the need for path monitoring, history, or a-priori knowledge of any kind.

7 Acknowledgments

We wish to thank Brian Youngstrom and the members of PlanetLab support who helped us with our trace, and Stavan Parikh who helped us develop SOSR. We also gratefully acknowledge the valuable feedback of Neil Spring, Ratul Mahajan, and Marianne Shaw, and we thank Miguel Castro for serving as our shepherd. This research was supported by the National Science Foundation under Grants ITR-0121341 and CCR-0085670 and by a gift from Intel Corporation.

References

[1] A. Akella, B. Maggs, S. Seshan, A. Shaikh, and R. Sitaraman. A measurement-based analysis of multihoming. In *Proceedings of ACM SIGCOMM 2003*, Karlsruhe, Germany, August 2003.

[2] D. G. Andersen, N. Feamster, S. Bauer, and H. Balakrishnan. Resilient overlay networks. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, Marseille, France, November 2002.

[3] D. G. Andersen, A. C. Snoeren, and H. Balakrishnan. Best-path vs. multi-path overlay routing. In *Proceedings of ACM/USENIX Internet Measurement Conference 2003*, October 2003.

[4] V. Cardellini, E. Casalicchio, M. Colajanni, and P. S. Yu. The state of the art in locally distributed web-server systems. Technical Report RC22209 (W0110-048), IBM T. J. Watson Research Center, October 2001.

[5] M. Dahlin, B. Chandra, L. Gao, and A. Nayate. End-to-end wan service availability. *IEEE/ACM Transactions on Networking*, 11(2), April 2003.

[6] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, 12(5):662–675, May 1986.

[7] N. Feamster, D. Andersen, H. Balakrishnan, and F. Kaashoek. Measuring the effects of internet path faults on reactive routing. In *Proceedings of ACM SIGMETRICS 2003*, San Diego, CA, June 2003.

[8] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 78–91, Saint Malo, France, 1997.

[9] Network Working Group. The ip network address translator (nat). RFC 1631, May 1994.

[10] B. Krishnamurthy, C. Willis, and Y. Zhang. On the use and performance of content distribution networks. In *Proceedings of ACM SIGCOMM Internet Measurement Workshop 2001*, November 2001.

[11] D. R. Kuhn. Sources of failure in the public switched telephone networks. *IEEE Computer*, 30(4):31–36, April 1997.

[12] C. Labovitz, A. Ahuja, A. Abose, and F. Jahanian. An experimental study of delayed internet routing convergence. In *Proceedings of ACM SIGCOMM 2000*, Stockholm, Sweden, August 2000.

[13] C. Labovitz, A. Ahuja, and F. Jahanian. Experimental study of internet stability and backbone failures. In *Proceedings of the 29th International Symposium on Fault-Tolerant Computing*, pages 278–285, 1999.

[14] R. Mahajan, D. Wetherall, and T. Anderson. Understanding bgp misconfiguration. In *Proceedings of ACM SIGCOMM 2002*, pages 3–16, Pittsburgh, PA, 2002.

[15] M. Mitzenmacher. On the analysis of randomized load balancing schemes. Technical Report 1998-001, Digital Systems Research Center, February 1998.

[16] Netfilter. `http://www.netfilter.org`.

[17] V. Paxson. End-to-end routing behavior in the internet. *IEEE/ACM Transactions on Networking*, 5(5):601–615, October 1997.

[18] PlanetLab. `http://www.planet-lab.org`.

[19] RouteScience. `http://www.routescience.com`.

[20] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking (MMCN) 2002*, January 2002.

[21] S. Savage, A. Collins, E. Hoffman, J. Snell, and T. Anderson. The end-to-end effects of Internet path selection. In *Proceedings of ACM SIGCOMM 1999*, Cambridge, MA, September 1999.

[22] Sockeye. `http://www.sockeye.com`.

[23] L. Subramaniam, S. Agarwal, J. Rexford, and R. H. Katz. Characterizing the internet hierarchy from multiple vantage points. In *Proceedings of IEEE INFOCOM 2002*, New York, USA, June 2002.

[24] R. Teixeira, K. Marzullo, S. Savage, and G. M. Voelker. Characterizing and measuring path diversity of internet topologies. In *Proceedings of ACM SIGMETRICS 2003*, pages 304–305, San Diego, CA, 2003.

[25] Alexander Siumann Yip. NATRON: overlay routing to oblivious destinations. Master's thesis, 2002.

[26] Y. Zhang, V. Paxson, and S. Shenker. The stationarity of internet path properties: Routing, loss, and throughput. Technical report, ACIRI, May 2000.