Source-sensitive routing

Matthieu Boutier, Juliusz Chroboczek

Source-sensitive routing

Matthieu Boutier and Juliusz Chroboczek
Univ Paris Diderot, Laboratoire PPS
Sorbonne Paris Cite, PPS, UMR 7126, CNRS, F-75205 Paris, France

***Abstract*—Source-sensitive (or source address-dependent) routing is a routing technique where routing decisions depend on both the source and the destination address of a packet. Source-sensitive routing solves some difficult problems related to multihoming in some topologies, and is therefore a useful addition to the multihoming toolbox. In this paper, we describe the semantics of source-sensitive packet forwarding, and describe our implementation of a source-sensitive extension to the Babel routing protocol — to our knowledge, the first complete implementation of source-sensitive routing —, including a disambiguation algorithm that makes our implementation work over standard networking APIs. We further discuss interoperability between ordinary next-hop and source-sensitive routing. Our implementation has seen a moderate amount of deployment, notably as a testbed for the IETF Homenet working group.**

I. Introduction

The routing paradigm deployed on the Global Internet is next-hop routing. In next-hop routing, per-packet forwarding decisions are performed by examining a packet's destination address only, and mapping it to a next-hop router. Next-hop routing is a simple, well understood paradigm that works satisfactorily in a large number of cases.

The use of next-hop routing restricts the flexibility of the routing system in two ways. First, since a router only controls the next hop, a route $A \cdot B \cdot C \cdots Z$ can only be selected by the router A if its suffix $B \cdot C \cdots Z$ has already been selected by a neighbouring router B, which makes some forms of global optimisation difficult or impossible. Other routing paradigms, such as circuit switching, label switching and source routing, do not have this limitation. (Source-routing, in particular, has been proposed multiple times as a suitable routing paradigm for the Global Internet [11], but has been forbidden due to claimed security reasons [1]).

Second, the only decision criterion used by a router is the destination address: two packets with the same destination are always routed in the same manner. Yet, there are other data in the IP header that can reasonably be used for making a routing decision – the TOS octet, the IPv6 flow-id, and, of course, the source address.

We call *source-sensitive* routing the modest extension of classical next-hop routing where the forwarding decision is allowed to take into account the source of a packet in addition to its destination. Source-sensitive routing gives a modest amount of control over routing to the sending host, which can choose among potentially many routes by picking a specific source address. The higher layers (transport or application) are therefore able to choose a route using standard networking APIs (collecting the host's local addresses and binding a socket to a specific address). Unlike source routing, however, source-sensitive routing remains a hop-to-hop mechanism, and therefore leaves local forwarding decisions firmly in the control of the routers.

II. Applications

The main application of source-sensitive routing is implementation of *multihoming*.

A. Multihoming

A multihomed network is one that is connected to the Internet through two or more physical links. This is usually done in order to improve a network's fault tolerance, but can also be done in order to improve throughput or reduce cost.

Classically, multihoming is performed by assigning *Provider-Independent* addresses to the multihomed network and announcing them globally (in the *Default-Free Zone* (DFZ)) over the routing protocol. The dynamic nature of the routing protocol automatically provides for fault-tolerance; improvements in throughput and reductions in cost can be achieved by careful engineering of the routing protocol.

Unfortunately, classical multihoming does not scale well, and can only be deployed by large networks. Every multihomed prefix must be announced to all of the providers, a setup which is generally impossible to achieve for home and small business networks. What is more, every such prefix must appear in the DFZ, which is replicated across all of the backbone routers of the Internet.

Note that it is not in general possible to implement classical multihoming using a single "Provider-Dependent" prefix. If a network is connected to two providers A and B, a packet with a source-address in an address range allocated to A will usually not be accepted by B — B will treat it as a packet with a spoofed source address and discard it [8]. What is more, A's prefix will not be announced by B in the default-free zone, and hence destinations in A's prefix will not be reachable over the link to B.

B. Multihoming with multiple source addresses

Since announcing the same Provider-Dependent prefix to multiple ISPs is not possible, it is a natural proposition to announce multiple PD prefixes, one per provider. In this approach, every host is assigned multiple addresses, one per provider, and extra mechanisms are needed (i) to choose a suitable source and destination address for each packet, and (ii) to properly route each outgoing packet according to both its source and its destination. In a sense, using multiple addresses

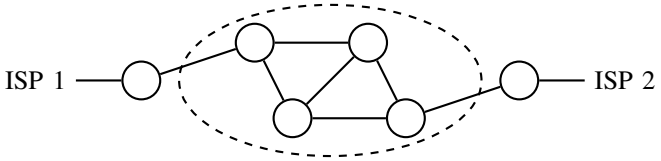
splits the difficult problem of multihoming into two simpler problems that are handled at different layers of the network stack.

1) Choosing addresses: The choice of source and destination addresses is typically left to the application layer. Typically, all destination addresses are stored within the DNS, and the sending host tries them all, either in turn [7] or in parallel [13]; similarly, all possible source addresses are tried in turn. Once a flow is established, it is no longer possible to change the source and destination addresses — from the user's point of view, all TCP connections are broken whenever a link outage forces a change of address. Address selection can be implemented in the operating system's kernel and libraries, or by the application itself, which is notably the case of most modern web browsers.

A different approach is to use a transport layer that has built-in support for multiple addresses and for dynamically renegotiating the set of source and destination addresses. One such transport layer is MPTCP [10]; we describe our experiences with MPTCP in Section VII-B.

2) Source-sensitive routing: As mentioned above, a provider will discard packets with a source address that is in a different provider's prefix. In a network that is connected to multiple providers, each outgoing packet must therefore be routed through the link corresponding to its source address.

When all the outgoing links are all connected to a single router, it is feasible to set up traffic engineering rules to ensure that this happens. There can be good reasons, however, why it is desirable to connect each provider to a different router: avoiding a single point of failure, load balancing, or simply that the various links use different link technologies that are not available in a single piece of hardware. In a home networking environment, the edge routers might be provided by the various service providers, with no possibility to consolidate their functionality in a single device.



With multiple edge routers, it is necessary that the routing protocol itself be able to route according to source addresses. We say that a routing protocol is *source-sensitive* when it is able to take both source and destination addresses into account in its routing decisions.

C. Other applications

In addition to multihoming with multiple addresses, we are aware of two problematic networking problems that source-sensitive routing solves cleanly and elegantly.

1) Overlay networks: Tunnels and VPNs are commonly used to establish a network-layer topology that is different from the physical topology, notably for security reasons. In many tunnel or VPN deployments, the end network uses its native default route, and only routes some set of prefixes through the tunnel or VPN.

In some deployments, however, the default route points at the tunnel. If this is done naively, the network stack attempts to route the encapsulated packets through the tunnel itself, which causes the tunnel to break. Many workarounds are possible, the simplest being to point a host route towards the tunnel endpoint through the native interface.

Source-sensitive routing provides a clean solution to that problem. The native default route is kept unchanged, while a source-sensitive default route is installed through the tunnel. The source-sensitive route being more specific than the native default route, packets from the user network are routed through the tunnel, while the encapsulated packets sourced at the edge router follow the native, non-specific route.

2) Controlled anycast: Anycast is a technique by which a single destination address is used to represent multiple network endpoints. A packet destined to an anycast address is routed to whichever endpoint is nearest to the source according to the routing protocol's metric. Anycast is useful for load balancing — for example, the global DNS root servers are each multiple physical servers, represented by a single anycast address.

For most applications of anycast, all of the endpoints are identical and it does not matter which endpoint is accessed by a given client. Some applications, however, require that a given user population access a well-defined endpoint — for example, in a Content Distribution Network (CDN), a provider might not want to serve nodes that are not its customers. Ensuring that this is the case by tweaking the routing protocol's metric (or "prepending" in BGP parlance) is fragile and error-prone.

Source-sensitive routing provides an elegant solution to this problem. With source-sensitive routing, each instance of the distributed server is announced using a source-sensitive route, and will therefore only receive packets from a given network prefix.

III. Related work

Multihoming is a difficult problem, and, unsurprisingly, there are many techniques available to implement it, none of which are fully general. In addition to classical network-layer multihoming, already mentioned above, there are a number of lower-layer techniques, the use of which is usually completely transparent to the network layer; we are aware of *Multi-Link PPP* [12], of *Ethernet link aggregation* (*port trunking*), of the use of MPLS to provide multiple paths across a rich link layer, as well as of proprietary techniques used by vendors of cable modems. Since these techniques work at the link layer, they are usually restricted to multihoming with a single provider.

All of these techniques are compatible, in the sense that they can be used at the same time. We imagine a home network where source-sensitive routing is used to access two providers, each of which is classicaly multihomed, over links that consist of multiple physical links combined at the link layer.

Source-sensitive routing itself is not a new idea [4], and implementing it manually on a single router using traffic

engineering interfaces is a well-documented technique [9]. Implementing source-sensitive routing within the routing protocol has been proposed by Bagnulo et al. [2], but the techniques used differ significantly from ours. First, the authors only deal with the non-overlapping case — where the different possible sources are disjoint —, which avoids the need for the disambiguation algorithm which is one of our main concerns. Second, they use a more general facility of an existing routing protocol rather than explicitly implementing source-sensitive routing. We find our more direct approach to be more intuitive, and expect it to be more reliable, since it doesn't require out-of-band agreement on the meaning of the labels carried by the routing protocol.

More generally, there are other applications of routing based on more information from the packet header than just the destination address. The traffic-engineering community has been experimenting with with routing based on the TOS octet of the IPv4 header for many years, and ability to do that is part of the OSPFv2 protocol. TOS-based routing is somewhat analoguous to source-sensitive routing, and many of the issues raised are similar; both could be seen as particular cases of "multi-dimensional routing".

Equal Cost Multipath (ECMP) is somewhat different. A router performing ECMP has multiple routes to the same destination, and chooses among them according to the value of a hash of the packet header. While ECMP does route on multiple header fields, the choice of fields used to choose a route in ECMP is a purely local matter, and does not need to be carried by the routing protocol.

IV. Source-sensitive routing

A. Next-hop routing tables

Ordinary next-hop routing consists in mapping a destination address to a next-hop. Obviously, it is not practical to maintain a mapping for each possible destination address, so the mapping table must be compressed in some manner. The standard compressed data structure is the *routing table* (or *Forwarding Information Base*, FIB), which ranges over *prefixes*, ranges of addresses the size of which is a power of two. The routing table can be constructed manually, but is usually populated by a routing protocol.

Since prefixes can overlap, the routing table is an ambiguous data structure: a packet's destination address can match multiple routing entries. This ambiguity is resolved by the so-called *longest-prefix rule*: when multiple routing table entries match a given destination address, the most specific matching entry is the one that is used.

More precisely, a prefix is a pair $P = p/plen$, where p is the first address in the prefix and $plen$ is the *prefix length*. An address d is in P when the first $plen$ bits of d match the first $plen$ bits of p. We say that a prefix $P = p/plen$ is more specific than a prefix $P' = p'/plen'$, written $P \leq P'$, when the set of addresses in P is included in the set of addresses in P'. Clearly, $P \leq P'$ if and only if $plen \geq plen'$, and the first $plen'$ bits of p and p' match.

The specificity ordering defined above has an important property: it is a *tree*, in the sense that given two prefixes P and P', they are either disjoint ($P \cap P' = \emptyset$), or one is more specific than the other ($P \leq P'$ or $P' \leq P$).

A routing table is a set of pairs (P, NH), where P is a prefix and NH, the *next hop*, is a pair of an interface and a (link-local) address; we further require that all the prefixes in a routing table be distinct. Since the set of prefixes is a tree, given an address d, either the set of prefixes in the routing table containing d is empty, or it is a chain (a totally ordered set); hence, there exists a most specific prefix P in the routing table containing d. The longest-prefix rule specifies that the next hop chosen for routing a packet with destination d is the one corresponding to this most specific prefix, if any.

B. Source-sensitive routing tables

Source-sensitive routing is an extension to next-hop routing where both the destination and the source of a packet can be used to perform a routing decision. Source-sensitive routers use a *source-sensitive* routing table, which is a set of triples (D, S, NH), where D is a destination prefix, S a source prefix, and NH is a next hop. Such an entry matches a packet with destination address d and source address s if d is in D and s is in S (note the ordering — destination comes first).

The specificity ordering generalises easily to pairs: a pair of prefixes (D, S) is more specific than a pair (D', S') when all pairs of addresses (d, s) which are in (D, S) are also in (D', S'); clearly, $(D, S) \leq (D', S')$ when $D \leq D'$ and $S \leq S'$.

Unfortunately, the set of destination-source pairs of prefixes equipped with the specificity ordering is no longer a tree. Consider the pairs $(2001{:}db8{:}1{:}/48, ::/0)$ and $(::/0, 2001{:}db8{:}2{::}/48)$. Clearly, these two pairs are not disjoint (the pair of addresses $(2001{:}db8{:}1{::}1, 2001{:}db8{:}2{::}1)$ is matched by both), but neither is one more specific than the other – the pair $(2001{:}db8{:}1{::}1, 2001{:}db8{:}3{::}1)$ is matched by the first but not the second, and, symmetrically, the pair $(2001{:}db8{:}4{::}1, 2001{:}db8{:}2{::}1)$ is matched by just the second. From a practical point of view, this means that a source-sensitive routing table can contain multiple most-specific entries, and thus fail to unambiguously specify a forwarding behaviour.

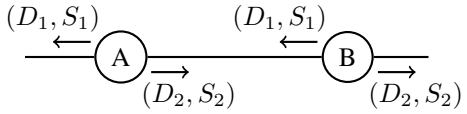
We say that a source-sensitive routing table is *ambiguous* when it contains multiple non-disjoint most-specific entries. Two entries that are neither disjoint nor ordered are said to be *conflicting*, and the set of (d,s) pairs that are matched by both entries is called a *conflict zone*. We note $A_1 \# A_2$ two entries in conflict.

C. Forwarding behaviour

In the presence of an ambiguous routing table, there exist packets that are matched by distinct most-specific entries. An arbitrary choice must be made in order to decide how to route such a packet.

Let us first remark that all routers in a single routing domain must make a consistent choice — having different routers

follow different policies within conflict zones may lead to persistent routing loops. Consider the following topology, with two source-sensitive routes indexed by the pairs (D_1, S_1) and (D_2, S_2) respectively, where packets matching (D_1, S_1) are sent towards the left of the diagram, and packets matching (D_2, S_2) are sent towards the right. If the two pairs are in conflict, and router A chooses (D_2, S_2) while B chooses (D_1, S_1), then a packet matching both pairs will loop between A and B indefinitely.



It is therefore necessary to choose a disambiguation rule that is uniform across the routing domain — any refinement of the specificity ordering that makes the set of pairs of prefixes into a tree will do. There are two natural choices: discriminating on the destination first, and comparing sources if destinations are equal, or discriminating on source first. More precisely, the destination-first ordering is defined by:

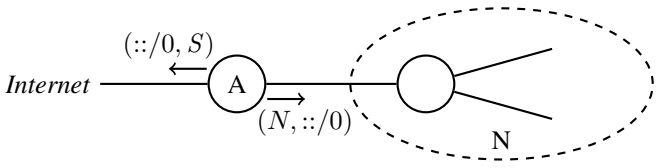
$$(D, S) \preceq (D', S') \text{ if } D < D' \text{ or } D = D' \text{ and } S \leq S',$$

while the source-first ordering is defined by

$$(S, S) \preceq_s (D', D') \text{ if } S < S' \text{ or } S = S' \text{ and } D \leq D'.$$

These orderings are isomorphic — hence, there is no theoretical argument that allows us to choose between them. An engineering choice must be made, based on usefulness alone.

The current consensus is that the destination-first ordering is the most useful of the two. Consider the following (fairly realistic) topology, where an edge router A announces a source-sensitive route towards the Internet, and a stub network N announces a (non-sensitive) route to itself. A packet matching both routes must follow the route towards N, since it is obviously the only route that can reach the destination, which implies that A must use the destination-first ordering. On the other hand, no such compelling examples of the usefulness of the source-first ordering are known to us.



In the following sections, we describe our experience with source-sensitive routing using the destination-first ordering. However, nothing in this article depends on the particular ordering being used, and our techniques would apply just as well to any algebraic tree that is a refinement of the specificity ordering.

V. Implementing source-sensitive routing

In the previous sections, we have described source-sensitive routing and shown how all routers in a routing domain must make the same choices with respect to ambiguous routing tables, and have argued in favour of the destination-first semantics. Whichever particular choice is made by an implementation of a routing protocol, however, must be implementable in terms of the primitives made available by the lower layers (the operating system kernel and the hardware).

In this section, we describe our experience with implementing a source-sensitive routing protocol. We first describe our experience with the native support for source-sensitive routing provided by recent Linux kernels (Section V-A). We then describe our "disambiguation" algorithm (Section V-B), which can be used to implement destination-first source-sensitive routing whatever ordering is used by the lower layers, as long as it is compatible with the specificity ordering, and which we use when implementing source-sensitive routing over the traffic engineering facilities of older Linux kernels.

A. Native source-sensitive FIB

Ideally, we would like the lower layers of the system (the OS kernel, the line cards, etc.) to implement destination-first source-sensitive routing tables out of the box. Such native support for source-sensitive routing is preferable to the algorithm described below, since no additional routes will need to be installed. In practice, however, while many systems have a facility for source-sensitive traffic engineering, this lower-layer support often has a behaviour different from the one that we require.

The Linux kernel, when compiled with the relevant options ("ipv6-subtrees"), claims to support source-sensitive FIBs natively, albeit for IPv6 only. Unfortunately, we found the support to be buggy — source-sensitive routes were treated as unreachable routes. This has been fixed since Linux 3.11, and the *netlink* interface is now able to accept a source-sensitive route and implements the destination-first behaviour; our implementation uses this interface for IPv6 if it is found to be present. In the case of IPv4, on the other hand, the "source" datum is silently ignored by *netlink*, and other techniques must be used.

B. Disambiguation of a routing table

All versions of Linux, at least some versions of FreeBSD, and probably other networking stacks, implement a facility to manipulate multiple routing tables and to select a particular one depending on the source address of a packet. Since the table is selected before the destination address is examined, this API implements the source-first behaviour, which is not what we aim to implement.

In this section, we describe a disambiguation algorithm that can be used to maintain a routing table that is free of ambiguities, and will therefore yield the same behaviour as long as the underlying forwarding mechanism implements a behaviour that is compatible with the specificity ordering (Section IV-B). All the forwarding mechanisms known to us satisfy this very mild hypothesis.

Recall that a routing table is ambiguous if there exists a packet that is matched by at least one entry in the table and such that there is no most-specific entry among the matching

entries. A necessary and sufficient property for a routing table to be non-ambiguous is that every conflict zone is equal to the union of more specific route entries.

The algorithm that we propose maintains, for each conflict, exactly one route entry that covers exactly the conflict zone. While a more parsimonious solution would be possible in some cases, it would greatly complicate the algorithm.

a) Weak completeness: We say that a routing table is *weakly complete* if each conflict zone is covered by more specific entries. More formally, T is weakly complete if $\forall r_1, r_2 \in T, r_1 \cap r_2 = \bigcup\{r \in T | r \leq r_1 \cap r_2\}$.

Theorem 1. *A routing table is non-ambiguous if and only if it is weakly complete.*

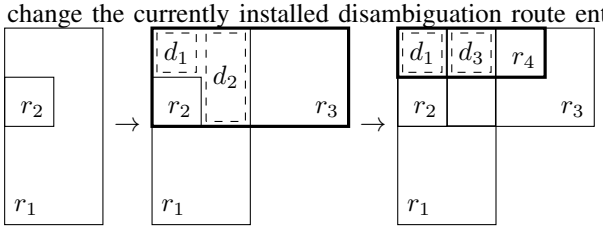
Proof: Let $U_x^y = \bigcup\{r \in T | r \leq x \cap y\}$. We need to show that T is non-ambiguous iff $\forall r_1, r_2 \in T, r_1 \cap r_2 = U_{r_1}^{r_2}$.

($\Leftarrow$) Suppose T is weakly complete, and consider two route entries $x, y \in T$ in conflict. By weak completeness, $U_x^y = x \cap y$, so for all addresses $a \in x \cap y$, there exists a route $r \in U_x^y$ such that $a \in U_x^y$. Since $r \in x \cap y$, we have $r < x$ and $r < y$, and r is more specific than $x \cap y$. Since this is true for all conflicts, the table is not ambiguous.

($\Rightarrow$) Suppose T is non-ambiguous and not weakly complete. Then there exist two entries $x, y \in T$ in conflict such that $x \cap y \neq U_x^y$. Consider an address $a \in x \cap y \setminus U_x^y$, and an entry $r \in T$ matching a. Clearly, $r \supsetneq x \cap y$, and so either $r \# x$ or $r \# y$, or $r > x$ and $r > y$. In all cases, r is not more specific than both x and y, so there is no minimum for the set of entries matching a. This contradicts the hypothesis, so if T is not ambiguous, it is weakly complete. ∎

Disambiguation with weak completeness is not convenient, since it may require adding multiple route entries to solve a single conflict, and the disambiguation routes added may generate additional conflicts. Suppose for example that the FIB first contains two entries $r_1 > r_2$, and we add $r_3 > r_2$ which conflicts with r_1 (see figure below). Since $r_2 < r_3$, there is no conflict within r_2, but we need disambiguation routes d_1 and d_2. The FIB is now weakly complete.

Suppose now that we add $r_4 < r_3$ in conflict both with r_1 and the disambiguation route d_2. We install a new disambiguation entry d_3. Note also that since $r_4 < r_3$, we need to use the next-hop of r_4 for the former region covered by d_1: we need to change the currently installed disambiguation route entry.



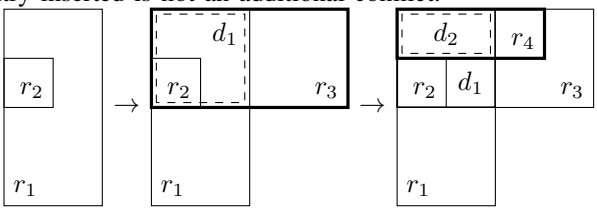
Some of this complexity can be avoided by requiring a stronger notion of completeness.

b) Completeness: A routing table is *(strongly) complete* if each conflict zone is covered by one route entry. More formally, T is complete if $\forall r_1, r_2 \in T, r_1 \cap r_2 \in T$. This obviously implies weak-completeness, and therefore a complete routing table is not ambiguous.

Our algorithm maintains the completeness of the routing table. An important property of completeness is that adding routes to achieve completeness does not lead to *another* conflict.

Proof: Suppose that $r_1 = (d_1, s_1)$ and $r_2 = (d_2, s_2)$ are two route entries in conflict, where $d_1 < d_2$ and $s_1 > s_2$. Consider the disambiguation entry $r_{sol} = (d_1, s_2)$ which disambiguates this conflict. Suppose now that r_{sol} is in conflict with another route entry $r_3 = (d_3, s_3)$. We have either $d_1 < d_3$ and $s_1 > s_2 > s_3$, in which case $r_3 \# r_1$; or $d_2 > d_1 > d_3$ and $s_2 < s_3$, in which case $r_3 \# r_2$. In either case, the conflict existed beforehand, and must therefore already have been resolved. ∎

Take the previous example again. When adding r_3, we add one route entry to cover the area d_1 ($r_1 \cap r_3$). Since r_2 is more specific, the new route entry does not affect the routing decision for addresses in r_2. When adding r_4, it is in conflict with both r_1 and the disambiguation route d_1, but for the same conflict zone $r_4 \cap r_1$. In that sense, the disambiguation route entry inserted is not an additional conflict.



c) Disambiguation routes: Disambiguation route entries do not appear on the wire, and in our implementation are not even inserted into the routing protocol's database; they are computed and inserted into the FIB on the fly, at route selection time. From the point of view of the routing protocol, disambiguation routes are a lower level implementation detail. Interestingly enough, we do not need to maintain a list of disambiguation routes that we have installed: when removing a route from the FIB, the set of disambiguation routes that need to be removed can be computed on the fly, similarly to what happens during route insertion.

The algorithm presented here is fully general, and can be generalised to different disambiguation orderings. We write $\preceq$ for the desired disambiguation ordering, in our case the lexicographic ordering on (d, s) pairs:

$$(d, s) \preceq (d', s') \quad \text{when} \quad d < d' \quad \text{or} \quad d = d' \text{ and } s \leq s'$$

d) Relevant conflicts: Consider a route entry R, and a set E of routing entries in conflict with R for the same conflict zone; all of these conflicts will have the same resolution. Moreover, if the resolution was caused by a route in E, then that was necessarily the most specific of the entries in E. Note that the minimum exists because elements of E have either the same destination, either the same source, and match at least one address in R.

Given a route entry r, we define the equivalence $\sim_r$ by $r_1 \sim_r r_2 \Leftrightarrow r_1 \cap r = r_2 \cap r$, i.e. two route entries are equivalent for $\sim_r$ if they have the same intersection with r. If two

equivalent route entries are in conflict with r, this means that they have the same conflict zone.

Quotienting a set of routing entries in conflict with r with this equivalence, and taking the minimum of each of the class of equivalence gives us exactly the routes that we care about.

1) Adding a route entry: Installing a new route entry in the FIB may make it ambiguous. For this reason, we must install the most specific routing entries first. In particular, we must install disambiguation entries before we install the route itself.

Let R be the route to install, and C the set of route entries in conflict with R, for which there is no natural solution, i.e. $C = \{\mathrm{R}' \in T | \mathrm{R}' \# \mathrm{R} \text{ and } \mathrm{R}' \cap \mathrm{R} \notin T\}$. We divide this set into two subsets, by conflict type, with only the relevant conflicts: $C_\prec = \{\min(E) | E \in (\{\mathrm{R}' \in C | \mathrm{R}' \prec \mathrm{R}\}/_{\sim_\mathrm{R}})\}$ and $C_\succ = \{\min(E) | E \in (\{\mathrm{R}' \in C | \mathrm{R}' \succ \mathrm{R}\}/_{\sim_\mathrm{R}})\}$.

For each route entry $\mathrm{R}_1 \in C_\succ$ (considering the most specific first), we first search, if it exists, the minimum route entry R_2 such that $\mathrm{R}_2 \# \mathrm{R}_1$ and $\mathrm{R}_2 \cap \mathrm{R}_1 = \mathrm{R} \cap \mathrm{R}_1$. If R_2 exists, then a disambiguation route is installed for that conflict, with NH_2 as next-hop: if $\mathrm{R} < \mathrm{R}_2$, then we must replace this next-hop by NH, otherwise the installed solution is the right one. If R_2 does not exist, we must add $((\mathrm{R}_1 \cap \mathrm{R}), NH)$ in the FIB.

For each route entry $\mathrm{R}_1 \in C_\prec$ (considering the most specific first), if there exists a route entry R_2 such that $\mathrm{R}_2 \# \mathrm{R}_1$ and $\mathrm{R}_2 \cap \mathrm{R}_1 = \mathrm{R} \cap \mathrm{R}_1$, then there is already the rigth disambiguation route installed. Otherwise, we must add $((\mathrm{R}_1 \cap \mathrm{R}), NH_1)$ in the FIB.

Finally, we must search if there exists two route entries in conflict for the zone of R. In that case, a disambiguation route entry has been installed, so R must replace it. Otherwise, R can be added normally. We end the procedure by adding R in our local RIB.

2) Removing a route entry: This time, we must first remove the less specific route first to keep the routing table unambiguous. Again, we write R for the route to be removed. First, remove R from the RIB. As for the addition, perhaps R is solving a conflict, in which case we cannot just remove it, but must first search for the entry covering that conflict, and replace R's next-hop. Otherwise, we just remove R from the FIB.

We consider $C_\prec$ and $C_\succ$ as previously defined.

For each route entry $\mathrm{R}_1 \in C_\succ$ (considering the less specific first), we first search, as we did for the adding process, for the minimum route entry R_2 such that $\mathrm{R}_2 \# \mathrm{R}_1$ and $\mathrm{R}_2 \cap \mathrm{R}_1 = \mathrm{R} \cap \mathrm{R}_1$. If R_2 exists and is more specific than R, there is nothing to do: the next-hop installed for this conflict is R_2. If it exists but is less specific than R, then NH is currently installed as a next-hop in the FIB, and must be change for NH_2. If R_2 doesn't exists, we must remove $((\mathrm{R}_1 \cap \mathrm{R}), NH)$ from the FIB.

For each route entry $\mathrm{R}_1 \in C_\prec$ (considering the less specific first), if there exists a route entry R_2 such that $\mathrm{R}_2 \# \mathrm{R}_1$ and $\mathrm{R}_2 \cap \mathrm{R}_1 = \mathrm{R} \cap \mathrm{R}_1$, then we must keep the disambiguation route entry in the FIB. Otherwise, we remove $((\mathrm{R}_1 \cap \mathrm{R}), NH_1)$.

3) Changing a route entry: This is the simplest case, since disambiguation routes must be maintained, and changed only if the route that we want to change has been selected for disambiguation. We can change first the disambiguation routes, or the route itself. Let R the route entry to change by R_{new}. We only consider $C_\succ$, as previously defined.

For each route entry $\mathrm{R}_1 \in C_\succ$, we search for the minimum route entry R_2 such that $\mathrm{R}_2 \# \mathrm{R}_1$ and $\mathrm{R}_2 \cap \mathrm{R}_1 = \mathrm{R} \cap \mathrm{R}_1$. If R_2 is R, then we replace $((\mathrm{R}_1 \cap \mathrm{R}), NH)$ by $((\mathrm{R}_1 \cap \mathrm{R}), NH_{new})$. Finally, we replace R by R_{new}.

C. External changes to the routing table

In the description above, we have asssumed that only our algorithm ever needs to manipulate the routing table. In practice, however, the routing table is also manipulated by other agents — other routing protocols or human operators. In principle, the same algorithm should be applied to externally changed routes; however, this is not implemented yet.

VI. Source-sensitive Bellman-Ford

The distributed Bellman-Ford algorithm is the foundation of a number of more or less widely deployed routing protocols, such as the venerable RIP, EIGRP, Babel and, to a certain extent, BGP and the inter-area sub-protocol of OLSR. In order to experiment with source-sensitive routing in a realistic manner, we have implemented a source-specific variant of the Babel routing protocol [6]. Our implementation has seen a moderate amount of deployment, most notably as a testbed for the IETF Homenet working group [5].

Roughly speaking, the source-specific extension to Babel extends Babel's routing table to be indexed by destination-source pairs rather than just destination prefixes. A new kind of update has been defined, that is able to carry both a source and a destination prefix. Since Babel's loop avoidance mechanism makes use of explicit requests, we have also created two new kinds of source-sensitive requests that mirror the existing requests in the unextended Babel protocol. Our implementation interoperates with unextended Babel routers, and does not interfere with the other existing extensions to the Babel protocol.

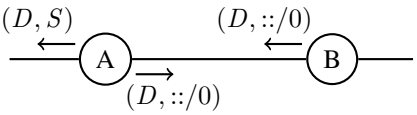
A. Interoperability

The Babel protocol has seen a moderate amount of deployment in production networks, and is usually deployed within cheap routers that can be difficult to update with a source-sensitive version of the protocol. We have therefore paid particular attention to the issue of interoperability between routers running the source-sensitive and unextended protocols.

An non-specific update having only the destination prefix (D) can be seen as a source-sensitive update for $(D, ::/0)$. Therefore, source-sensitive routers interpret non-sensitive updates as source-sensitive updates with a $::/0$ source prefix. Conversely, source-sensitive routers never send updates of the form $(D, ::/0)$, sending a non-specific update instead.

A more difficult issue is how a non-sensitive router should interpret a source-sensitive update. There are two possibilities: the source can be discarded and the update treated as non-specific, or the entire update can be discarded. The first of these possibilities can cause persistent routing loops.

Consider two nodes A and B, with A source-sensitive announcing a route to (D, S) (with $S \neq ::/0$). When B receives the announcement, it ignores the source information, installs and announces it as D. This is reannounced to A, which treats it as $(D, ::/0)$. Packets destined to D but not sourced in S will be forwarded by A to B, and by B to A.



If non-source-sensitive nodes rejects source-sensitive updates, but source-sensitive nodes accept non-source-sensitive updates with $::/0$ source, then source-sensitive nodes can communicate entries of the form $(D, ::/0)$ as (D), and are completely compatible with non-source-sensitive nodes. In this case, Bellman-Ford will eventually converge to a loop-free configuration.

In general, discarding source-sensitive routes by non-sensitive will cause routing blackholes. Intuitively, unless there are enough non-specific routes in the network, non-sensitive routers will have to discard packets in some cases. A simple sufficient condition for avoiding blackholes is to build a connected source-sensitive backbone including all the edge routers, and announce a default route towards the backbone.

B. Implementation details

There are two natural ways to encode source-sensitive updates and requests within the framework of Babel's extension mechanism: by defining a new set of TLVs, or by adding a sub-TLV to existing TLVs. We have defined a new set of TLVs, since these will be ignored by existing implementations of Babel; using a sub-TLV would cause just the sub-TLV to be ignored, which, as we have seen above, could cause persistent routing loops.

The standalone implementation of Babel has an extensive framework for redistribution and filtering. We have extended this framework to allow a redistribution filter to attach a source to a redistributed route. While this can cause persistent routing loops to occur, this is not unusual with redistribution.

In order to allow traditional Babel nodes to participate to multihomed networks, we have added an option allowing a source-sensitive Babel node to map source-sensitive updates (D, S) to both a source-sensitive update and a non-specific update for D while rejecting all updates for D. This is clearly an unsafe hack, which is safe only if all source-sensitive *routers* have this option activated (or employ filtering); however, we have found that it greatly simplified the administration of our experimental network.

VII. Experimental results

We have implemented both schemes described in Sections V-A and V-B within `babeld`, a Linux implementation of Babel [6], a distance-vector protocol based on a loop-free variant of the Bellman-Ford algorithm. This has allowed us to perform a number of experiments which we describe in this section.

```
# ip rule show
0: from all lookup local
101: from 192.168.4.0/24 lookup 11
32766: from all lookup main
32767: from all lookup default
# ip route show
default via 172.23.47.254 dev eth1 proto static
172.23.32.0/20 dev eth1 proto kernel src 172.23.36.138
192.168.4.20 via 192.168.4.20 dev tun-ariane proto 42 onlink
192.168.4.30 via 192.168.4.30 dev wlan1 proto 42 onlink
[...]
# ip route show table 11
default via 192.168.4.20 dev tun-ariane proto 42 onlink
192.168.4.20 via 192.168.4.20 dev tun-ariane proto 42 onlink
192.168.4.30 via 192.168.4.30 dev wlan1 proto 42 onlink
[...]
```

Fig. 1. IPv4 routing table on a router using a VPN

Our experimental network consists of a mesh network consisting of a dozen OpenWRT routers and a single server running Debian Linux. Two of the mesh routers have a wired connection to the Internet, and are connected to the server through VPNs (over IPv4). All of the routers run our modified version of the Babel protocol.

IPv4 connectivity for the mesh is provided by the Debian server, which acts as a NAT box. The IPv6 connectivity is more interesting: there are two IPv6 prefixes, one of which is a native prefix provided by our employer's network, the other one being a prefix specific to the server and routed through the VPN. The network therefore has two source-sensitive default IPv6 routes.

A. Routing table, and VPN connectivity

Figure 1 shows an excerpt of the routing tables of one of the two wired routers. The modified `babeld` daemon has allocated a non-default routing table, table 11, and inserted routes (marked as `proto 42`) into both the default `main` table and table 11. The former table contains non-specific routes: the default route and the `/20` subnet learned by our laboratory's DHCP, and host routes to individual mesh nodes of our testbed. Note the VPN is built over this default route.

Table 11 contains routes for locally originated packets, sourced in 192.168.4.0/24. The only "real" route in this table is the default route, which prevents the VPN from attempting to "enter itself". The other routes are disambiguation routes, automatically generated by the algorithm described in Section V-B.

Routing table 11 is specific to addresses from our local network: the default route it contains is also specific to that network. The other routing entries we show are disambiguation entries, added by our algorithm so that packets from our local network to our local network will not leave the network by following the default route. These entries are copies of the one present in the main routing table.

According to the default route of table 11, packets destined to the Internet and from our local network are routed through our VPN. The encapsulated VPN packets, sourced in our

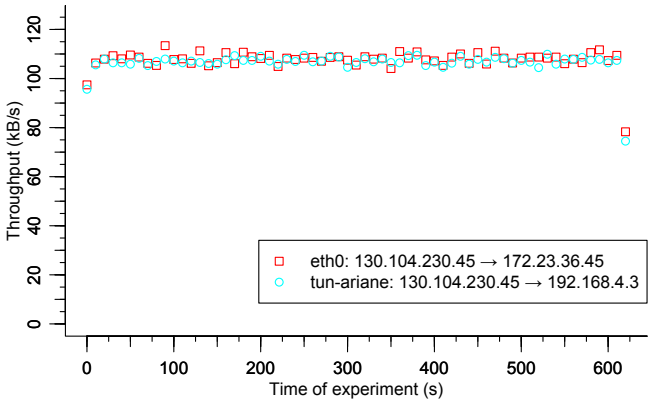


Fig. 2. Download using MPTCP and traffic control

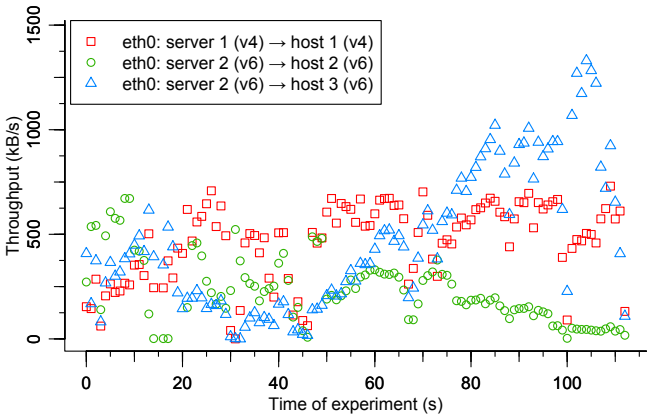


Fig. 3. Download using MPTCP

laboratory network, avoid table 11 and are routed by the main routing table's default route through the native network.

B. Multipath TCP

Multipath TCP [10] is an extension to TCP which multiplexes a single application-layer flow over multiple network layer sub-flows, and attempts to use as many distinct routes as possible, and to either carry traffic over the most efficient one or to perform load balancing. An obvious application would be a mobile node (a telephone) with permanent connectivity to a cellular network and intermittent WiFi connectivity. MPTCP is able to use the cellular link when WiFi is not available, and switch to WiFi when available without dropping already established connections.

Multipath TCP and source-sensitive routing turn out to be a surprisingly good match. MPTCP is able to use all of the addresses of the local host, and to dynamically probe the reliability and performance of packets sourced from each of those.

We have performed two tests that both consist in downloading a 110 MB file over MPTCP from the MPTCP website. In the first test (Figure 2), a desktop computer is directly connected to the source-senstively routed wired network, and is configured with two IPv4 addresses. The Linux `tc` subsystem is used to limit each of the addresses to 100 kB/s traffic; MPTCP is able to reliably download at 200 kB/s.

In the second test (Figure 3), a laptop's WiFi interface is configured with three addresses (one IPv4 and two IPv6). MPTCP multiplexes the traffic across the three routes, and balances their throughput dynamically.

VIII. Conclusion and Further Work

Source-sensitive routing is a modest extension to next-hop routing that keeps the forwarding decisions firmly within control of the routers while allowing end hosts a moderate and clearly defined amount of control over the choice of routes. Since source-sensitive routing can cause ambiguous routing tables, we have defined the behaviour that we believe source-sensitive routers should have, and shown how combining different behaviours in the same network can cause persistent routing loops. Similar care must be taken when combining non-sensitive with source-sensitive routers in the same network. We have proposed two ways to implement source-sensitive routing, and obtained experimental results that show that source-sensitive routing can be usefully exploited by the transport layer protocol MPTCP. Our implementation is of production quality, and has seen a modest amount of deployment, notably as a testbed for the ideas of the IETF Homenet working group.

While we enjoy working with distance-vector protocols, much of the networking community appears to have converged on using the OSPF protocol for internal routing. OSPF is a rich and complex protocol, and while many of our techniques should apply without difficulty to it, actually implementing a full source-sensitive variant of OSPF without sacrificing any of its flexibility remains a challenging endeavour.

It was a pleasant surprise to discover that unmodified MPTCP can use source-sensitive routes without any manual configuration. However, we claim that source-sensitive routing can also be exploited at the application layer without any changes to the transport layer; we are therefore considering modifying an application to make full use of multiple source addresses; interesting candidates include implementations of the BitTorrent file sharing protocol based on μTP, as well as the *Mosh* [14] remote shell replacement.

References

[1] J. Abley, P. Savola, and G. Neville-Neil. Deprecation of Type 0 Routing Headers in IPv6. RFC 5095, December 2007.

[2] Marcelo Bagnulo, Alberto García-Martínez, Juan Rodríguez, Arturo Azcorra. The Case for Source Address Dependent Routing in Multihoming. Quality of Service in the Emerging Networking Panorama. Lecture Notes in Computer Science Volume 3266, 2004, pp. 237-246.

[3] F. Baker. IPv6 Source/Destination Routing using OSPFv3. Internet-Draft draft-baker-ipv6-ospf-dst-src-routing-03, IETF Secretariat, August 2013.

[4] F. Baker and P. Savola. Ingress Filtering for Multihomed Networks. RFC 3704 and BCP 84, March 2004.

[5] T. Chown, Ed. IPv6 Home Networking Architecture Principles. Internet-Draft draft-ietf-homenet-arch-17. July 2014.

[6] J. Chroboczek. The Babel Routing Protocol. RFC 6126, April 2011.

[7] R. Draves. Default Address Selection for Internet Protocol version 6 (IPv6). RFC 3484, February 2003.

[8] P. Ferguson and D. Senie. Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing. RFC 2827 and BCP 38, May 2000.

[9] Bert Hubert et al. *Linux Advanced Routing and Traffic Control*. Available online at http://www.lartc.org/.

[10] Costin Raiciu, Christoph Paasch, Sébastien Barré, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP. In *USENIX Symposium of Networked Systems Design and Implementation (NSDI'12), San Jose (CA)*, 2012.

[11] Jerome H Saltzer, David P Reed, and David D Clark. Source routing for campus-wide internet transport. In *Proc. IFIP WG 6.4 International Workshop on Local Networks*, pages 1–23, 1980.

[12] K. Sklower, B. Lloyd, D. Carr and T. Corradetti. *The PPP Multilink Protocol (MP)*. RFC 1990, August 1996.

[13] D. Wing and A. Yourchenko. Happy Eyeballs: Success with Dual-Stack Hosts. RFC 6555, April 2012.

[14] Keith Winstein and Hari Balakrishnan. Mosh: An Interactive Remote Shell for Mobile Clients. In *USENIX Annual Technical Conference*, Boston, MA, June 2012.