Source-sensitive routing

Matthieu Boutier, Juliusz Chroboczek

Source-sensitive routing

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

ABSTRACT

Source-sensitive routing is a routing technique where routing decisions are made according to both the source and the destination address of a packet. This is a slight refinement of ordinary next-hop routing, as used in the Global Internet, where packets are routed according to their destination only. Source-sensitive solves a number of difficult problems in multihomed edge networks as well as in the presence of tunnels.

This paper describes our experience with the design and implementation of a source-sensitive extension to a distance-vector routing protocol (the Babel protocol). First, we define the behaviour of our source-sensitive routers, and show that mixing different behaviours in a single routing domain causes persistent routing loops. We then describe two implementation techniques for source-sensitive routing, and show how source-sensitive routers can interoperate with ordinary next-hop routers in a single routing domain. We conclude with experimental data obtained with our implementation.

1. INTRODUCTION

The main routing paradigm deployed on the Global Internet is next-hop routing. In next-hop routing, forwarding decisions are performed per-packet, and consist in examining a packet's destination address only, and mapping it to a next-hop router.

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 [SRC80]), but has been forbidden due to claimed security reasons [ASNN07].

Second, the only decision criterion used by a router is the destination address. This implies that two packets with the same destination are routed identically, which is not always desirable. 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.

Outline of this paper.

In Section 2 of this paper, we describe in more detail some of the applications of source-sensitive routing. In Section 3, we describe the structure of source-sensitive routing tables, and the related technical difficulties. In Section 4, we describe two implementation techniques for source-sensitive routing tables. In Section 5, we generalise the familiar Bellman-Ford family of routing algorithms to source-sensitive routing, and study the interoperability issues between the classical and the source-sensitive variants of the algorithm. Finally, in Section 6, we present some experimental results obtained with our experimental source-sensitive variant of the Babel routing protocol [Chr11].

2. APPLICATIONS

Source-sensitive routing allows a weak form of multihoming, particularly suited to multihomed edge networks where the upstream ISPs are not actively supporting multihoming. It also solves some accessibility problems in the presence of tunnels or VPNs.

2.1 Multihomed networks

A multihomed network is a network that is connected to two or more providers. There are two principal reasons for multihoming an edge network: reliability — a multihomed network doesn't lose its connectivity when one of its providers fails —, and performance — a multihomed net-

work is able to send its traffic through the fastest of many providers, or, in ideal conditions, to load-balance between multiple providers.

A multihomed network has multiple routes to each destination, one through each upstream; in particular, a multihomed network typically has multiple default routes.

Classical multihoming.

Classically, multihoming is performed by assigning a *provider-independent* (PI) addresses to the hosts in the multihomed network, and announcing and receiving routes over a dynamic routing protocol (typically BGP) to all of the upstream networks. Reliability is a natural consequence of the dynamic nature of the routing protocol, which will route around a failed upstream provider. Load-balancing can usually be achieved by manually tweaking route priorities, using mechanisms such as BGP *prepends*, the BGP *local preference* or the *MED*.

Multihoming with multiple source addresses.

Classical multihoming relies on the upstream providers accepting routes to the PI prefix and routing packets sourced within the PI prefix. Edge networks, however, and especially home networks, are typically provisioned with addresses originally allocated to one of the upstream providers — *provider dependent* (PD) addresses. The other providers will not typically accept routes to a foreign PD prefix; additinally, they will drop packets sourced within a foreign prefix (this is called *BCP-38 filtering* [FS00]).

The edge network multihoming issue can be solved by assigning multiple addresses to each host, one for each upstream ISP. A hosts chooses one of the exit routers by selecting one of its source addresses — hence the need for source-sensitive routing.

There are multiple places where the selection of the source address may happen. Ideally, the failure of one of the edge routers might cause the relevant addresses to be unassigned, for example because of an IPv6 route announcement or a DHCP lease timing out. More realistically, the upper layers could send probe traffic from each of the source addresses, and use an address that happens to work — this is done at the transport layer by MPTCP, which we describe in Section 6.2.

As to load-balancing, it could be achieved either by choosing source addresses randomly or by applying RFC 3484 rules. Again, MPTCP solves the problem quite nicely at the transport layer.

2.2 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-specific default route is installed through the tunnel. The source-specific 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.

3. SOURCE-SENSITIVE ROUTING

As mentioned before, in next-hop routing a next hop router is chosen by considering the destination address of a packet. This is implemented by using a data strcuture known as the *routing table* (or sometimes the *FIB*), which is conceptually a set of pairs (P, NH) where P is the description of a set of packets and NH the address of a next-hop router.

3.1 Classical next-hop routing tables

In classical next-hop routing, packets are forwarded considering their destination addresses only, so the packet pattern is a destination prefix, and the routing table a set of pairs $(\mathrm{D}, \mathrm{NH})$, where D is a destination prefix. However, a packet can match multiple routing entries, since an address can be included in multiple different prefixes. For example, the address 2001:DB8:0:1::4 is both in 2001:DB8::/56 and 2001:DB8:0:1::/64. In practice, we select the entry satisfying the *most specific prefix* rule.

A prefix can be seen as a the set of addresses which, truncated at the length of the prefix, are equal to the prefix. That is, a prefix p of length n is the set of all addresses a such that the first n bits of a are the same than the first n bits of p. Equipped with the inclusion ordering, the set of prefixes is a *tree*: any two prefixes p and p' are either disjoint or ordered. Hence, any set of prefixes of nonempty intersection is a chain (a total order), and therefore has a minimum element – the *most specific prefix*. The most specific prefix rule says that the routing table entry $(\mathrm{D}, \mathrm{NH})$ used for routing a prefix destined for a destination d is the one whose prefix is the most specific among the set of prefixes containing d within the routing table.

3.2 Source-sensitive routing tables

In source-sensitive routing, packets are forwarded considering both their destination and source addresses. The packet pattern is a pair (D, S), and the source-sensitive routing table a set of triples $(\mathrm{D}, \mathrm{S}, \mathrm{NH})$. As for classical next-hop, a packet can match multiple patterns. However, there is no obvious analogue to the most specific prefix rule, since the inclusion ordering is no longer a tree. For example, a packet $(2001{:}DB8{:}0{:}1{::}4, 2001{:}DB8{:}0{:}2{::}2)$ matches both $(2001{:}DB8{::}/56, 2001{:}DB8{:}0{:}2{::}/64)$ and $(2001{:}DB8{:}0{:}1{::}$

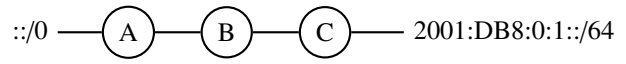
/64, 2001:*DB*8::/56), but neither of the two entries is more specific than the other.

We say that a routing table is *ambiguous* if it contains two patterns of non-empty intersection that are not comparable. We say that two such uncomparable routing entries are in *conflict*, and we call *conflict zone* the set of addresses matched by both theses entries.

3.3 Behaviour of routing with ambiguous tables

In the presence of ambiguity, multiple behaviours could be implemented by routers. For example, a router could choose the first installed matching entry, which is a kind of non-determinism, by the most specific source, or simply treat conflict zones as unreachable.

First, routers must have the exact same behaviour, lest persistent routing loops occur. Indeed, consider the following topology, and suppose that the routing table at B contains a source-sensitive default route through A, and a non-source-sensitive route towards 2001:DB8:0:1::/64 through C. If router B implements a "destination first" rule, while router C implements a "source first" rule, then C will send B's packets back to B, which in turn will send it back to C, etc.



A consequence of this observation is that no non-deterministic behaviour is allowable: the disambiguation rule must be designed to make the set of routes into a tree. Routing by destination or source first are the two natural linearisations of the ordering.

Consider again the previous example. The correct behaviour here is clearly to send a packet destined to 2001:*DB*8:0:1::/64 through C – this is the only choice that has a chance of getting the packet to the right destination. We have been unable to find any practical situation where a different behaviour would be desirable.

We therefore claim that the correct behaviour is to route by destination first. More formally, given two conflicting routing entries (D_1, S_1, NH_1) and (D_2, S_2, NH_2), a packet in the conflict zone is sent to NH_1 if and only if D_1 is strictly more specific than D_2, or D_1 and D_2 are equal and S_1 is more specific than S_2. It is easily checked that this rule makes the set of source-sensitive patterns into a tree.

4. DISAMBIGUATING ROUTING TABLES

Ideally, we would like the lower layers of the system (the OS kernel, the line cards, etc.) to implement source-sensitive routing tables out of the box, with the right disambiguation behaviour already present. 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.

For example, all recent versions of the Linux kernel have the ability to manipulate multiple routing tables, and to select a given routing table depending on the source address of a packet. Since the selection of the routing table happens before the destination address is considered, the behaviour that is implemented is that of source-first routing, as opposed to the destination-first routing that we advocate.

This section is structured as follows. We first describe the native API that exists in some Linux kernels; then we describe a disambiguation algorithm that, we claim, can be used to implement destination-first source-sensitive routing on any system that has some facility for source-sensitive routing, whatever its exact behaviour, as long as it is compatible with the pointwise ordering (Section 4).

4.1 Native source-sensitive FIB

In an ideal world, destination-first source-sensitive routing would be directly implemented by the lower layers (e.g. the OS kernel). Such native support for source-sensitive routing is preferable to the algorithm described below, since no additional routes will be installed in the FIB. Our investigations of such native support yielded disappointing results.

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 was fixed in Linux 3.11, and the *netlink* interface is now able to accept a source-sensitive route and implements the destination-first behaviour. In the case of IPv4, on the other hand, the "source" datum is silently ignored by *netlink*, and other techniques must be used.

All versions of Linux, and some versions of FreeBSD, implement the ability 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 — the algorithm described below is therefore necessary.

4.2 Disambiguation of a routing table

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 point-to-point ordering over pairs (D, S). All the forwarding mechanisms known to us satisfy this very mild hypothesis.

Recall that a routing 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 economic solution might be possible, it would appear to be overly complex.

Prefixes as set of addresses.

Recall that a prefix P of length n is a sequence of n bits $b_1b_2...b_n$. An address a matches P if the first n bits of a are equal to the n bits of P. A prefix P can be identified with the set of addresses that it matches. The inclusion relation on sets of addresses induces an ordering on prefixes; we say that a prefix P_1 is *more specific* than P_2, written $P_1 \leqslant P_2$, when the set of addresses matched by P_1 is a subset of the addresses matched by P_2.

An important property of prefixes is that they form a *tree*: given two prefixes P_1 and P_2, they are either disjoint (there is no address that they both match), or one is more specific than the other ($P_1 \leqslant P_2$ or $P_1 \leqslant P_2$).

Pairs of prefixes.

Taken pointwise, the partial order on prefixes induces the product ordering on pairs of prefixes. Given two pairs of prefixes $A_1 = (P_1, Q_1)$ and $A_2 = (P_2, Q_2)$, we write $A_1 \leqslant A_2$ if $P_1 \leqslant P_2$ and $Q_1 \leqslant Q_2$. Equipped with this ordering, the set of pairs of prefixes is not a tree; we say that A_1 *conflicts* with A_2, written $A_1 \# A_2$, when A_1 and A_2 are neither ordered nor disjoint.

A routing table is *ambiguous* if there exists a pair of addresses (d, s) that matches a set of entries among which there is no most specific one. Note that the only way for an address to not have a most specific matching entry is when it matches two most specific entries in conflict.

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 $\forall r_1, r_2 \in T, r_1 \cap r_2 = \bigcup\{r \in T | r \leqslant r_1 \cap r_2\}$.

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

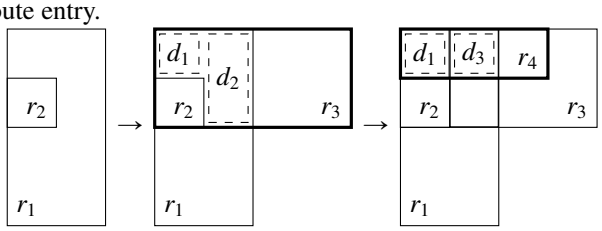
PROOF. Let $U_x^y = \bigcup\{r \in T | r \leqslant x \cap y\}$. We need to show that T is non-ambiguous iff $\forall r_1, r_2 \in T, r_1 \cap r_2 = U_{r_1}^{r_2}$.

($\Leftarrow$) 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 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.

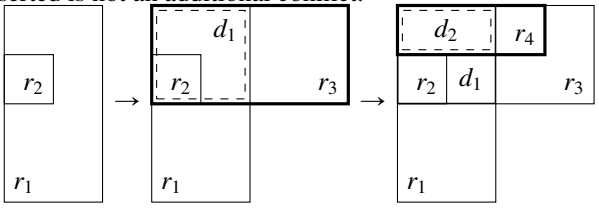
Completeness.

We say that 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 week-completeness, and therefore complete routing table is not ambiguous.

Our algorithm maintain 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 $r3$, 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 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.



Disambiguation routes.

Disambiguation route entries do not appear on the wire, and in our implementation are not even inserted into the RIB; 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 have found that 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:

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

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 more specific of the entries in E. Note that the minimum exist 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.

4.2.1 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 $\mathrm{R} \prec \mathrm{R}_2$, 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 $((\mathrm{R}_1 \cap \mathrm{R}), \mathrm{NH})$ 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 $\mathrm{R}_2 \cap \mathrm{R}_1 = \mathrm{R} \cap \mathrm{R}$, then there is already the rigth disambiguation route installed. Otherwise, we must add $((\mathrm{R}_1 \cap \mathrm{R}), \mathrm{NH}_1)$ 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.

4.2.2 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 $\mathrm{D}_2 = \mathrm{D}$ and $\mathrm{S}_2 > \mathrm{S}_1$. 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 $(\mathrm{D}, \mathrm{S}_1, \mathrm{NH})$ 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 $\mathrm{S}_2 = \mathrm{S}$ and $\mathrm{D}_2 > \mathrm{D}_1$, then we must keep the disambiguation route entry in the FIB. Otherwise, we remove $(\mathrm{D}_1, \mathrm{S}, \mathrm{NH}_1)$.

4.2.3 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 $C_\prec$, as previously defined.

For each route entry $\mathrm{R}_1 \in C_\succ$, if R is the minimum route entry having D as destination and such that $\mathrm{S}_2 > \mathrm{S}_1$, then we replace $(\mathrm{D}, \mathrm{S}_1, \mathrm{NH})$ by $(\mathrm{D}, \mathrm{S}_1, \mathrm{NH}_{new})$. Finally, we replace R by R_{new}.

4.3 External FIB changes

In the description above, we assume that only our algorithm ever manipulates the FIB. In practice, however, the FIB is manipulated by other agents — other routing protocols, or human operators. The same algorithm should be applied to externally changed routes[1].

[1] This is not currently implemented.

5. 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 Bellman-Ford, each node broadcasts to its neighbours the set of route identifiers that it can reach, with an associated cost. In traditional next-hop routing, identifiers are destination prefixes, and in source-sensitive next-hop routing, identifiers are pairs of destination and source prefixes.

In this section, we consider a network composed of both traditionnal and source-sensitive routers, and discuss how they can interoperate.

5.1 Interoperability

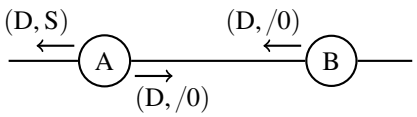
We expect a single routing area to contain both source-sensitive and traditional routers, either because source-sensitive routers are deployed incrementally, avoiding a flag day, or because some devices cannot be upgraded to implement full source-sensitive routing, for technical, economic or political reasons. For this reason, source-sensitive routers must be able to communicate with traditional routers.

An non-specific update having only the destination prefix (D) can be seen as a source-specific update for $(\mathrm{D}, /0)$. Therefore, source-sensitive routers should interpret non-sensitive updates as source-specific updates with a $/0$ source prefix. Conversely, non-sensitive routers should accept updates of the form $(\mathrm{D}, /0)$ and treat them as non-specific updates; this is most easily achieved by not sending specific updates with a /0 source, and sending a non-specific update instead.

A more difficult issue is how a non-sensitive router should interpret a source-specific update with a non-trivial source prefix. There are two possibilities: the source can be discarded and the update treated as non-specific, or the entire update can be discarded. As we shall see, the first of these possibilities doesn't work, while the latter does.

What doesn't work.

Discarding the source of a source-specific update and treating it as non-specific can cause persistent routing loops. Indeed, consider two nodes A and B, with A source-sensitive announcing a route to (D, S) (with $\mathrm{S} \neq /0$). 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 $(\mathrm{D}, /0)$. Packets destined to D but not sourced in S will be forwarded by A to B, and by B to A.



Discarding specific routes.

If non-source-sensitive nodes rejects source-sensitive updates, but source-sensitive nodes accept non-source-sensitive uptades with $/0$ source, then source-sensitive nodes can communicate entries of the form $(\mathrm{D}, /0)$ as (D), and are completely compatible with non-source-sensitive nodes.Since there is no identifier change, Bellman-Ford converges to a loop-free set of routes.

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.

5.2 Implementation details

Routing protocols must implement source-sensitiveness as an incompatible extension. In our implementation of the Babel routing protocol [Chr11], this is achieved by the introduction of a new TLV, which is silently ignored by other Babel nodes. Source-sensitive nodes continue understanding the previous TLV, and announce routes with $/0$ source as non-source-sensitive, i.e. with the previous TLV.

In our implementation, bootstrapping is achieved at redistribution time, by allowing a redistribution filter to map a non-specific route to a source-specific one. While this may cause routing loops in general, it is not unusual in routing protocols for careless redistribution to cause routing loops.

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-specific 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 simplified the administration of our network.

6. EXPERIMENTAL RESULTS

We have implemented both schemes described in Section 4 within `babeld`, our Linux implementation of Babel [Chr11], 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 desribe in this section.

Description of our testbed.

Our experimental network consists of a mesh network consisting of a dozen OpenWRT routers and a single Debian server. Two of the mesh routers have a wired connection to the Global 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 Debian box and routed

```
# 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  scope link  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
[...]
```

Figure 1: v4 routing table on a router using a VPN

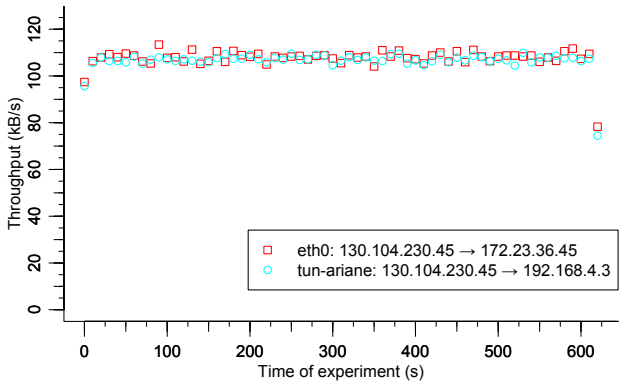


Figure 2: Download using MPTCP and traffic control

through the VPN. The network therefore has two source-specific default IPv6 routes.

6.1 Routing table, and VPN connectivity

Figure 1 shows an excerpt of the routing table 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, both to the `/20` announced by the Debian server and host routes to individual mesh nodes, as well as a default route through the VPN.

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 4.

The 11 routing table 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 such 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.

By the default route of the 11 routing table, packets destined to the Internet and from our local network are well routed through our VPN. The encapsulated VPN packets, sourced in our laboratory network, avoid table 11 and are routed by the main routing table's default route through our network laboratory.

6.2 Multipath TCP

Multipath TCP [RPB+12] 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 uses all of the addresses of the local host, and to dynamically probe the reliability and performance of packets sourced from each of those which made it particularly straightforward for source sensitive routing.

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 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.

7. RELATED WORK

Source-sensitive routing is somewhat related to TOS routing, as found for example in OSPFv2. In both cases, multiple routes are provided by the network layer, and upper layers have a limited choice of routes. In the case of the TOS routing, the higher layers set the TOS field to choose a particular metric to optimise (bandwidth, latency, etc.). In [AGKT98], the authors note that TOS routing [LHH95] attempts to improve network utilisation by providing multiple routes, similarly to what happens with source-sensitive routing. However, contrary to source-sensitive routing, measuring the relative performance of the different routes is the network layer's reposibility.

The first mention of source-sensitive routing that we are

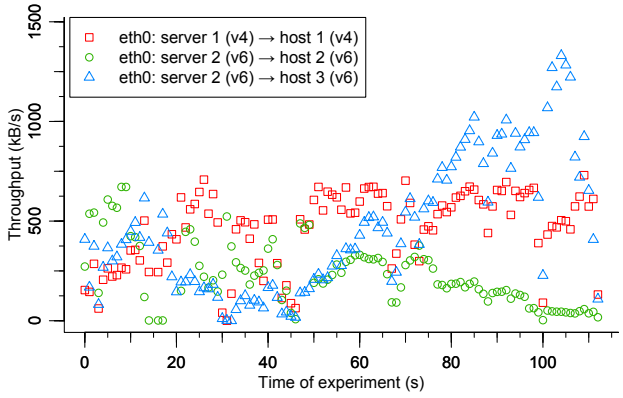


Figure 3: Download using MPTCP

aware of is from early 2004 [BS04]. The notion of ambiguity appears years later: in [Bak11], Baker notes that two entries of the routing table can match the same address without beeing comparable. The solution he proposes is to disambiguate using the route's metric: similar to our algorithm, he proposes adding an additional route entry at the intersection of the two incomparable entries, but chooses the next-hop of the lowest metric one.

In early 2013, four IETF Internet Drafts were published on the subject of source-sensitive routing. In the first version of [Bak13], Baker speaks about ambiguity of routing tables, this time without specifying a suggested behaviour. The last revision of the draft proposes the destination first behaviour, similar to the present paper.

Approximately at the same time, Troan et al. [TC13] also remarked that routing tables can be ambiguous, and proposed a behaviour that is equivalent to the destination-first behaviour of the present paper. Xu et al. [XYWW13] were more protocol oriented: they proposed an overview and some notes about how should be implemented a variant of OSPF for source-sensitive routing. They notice the problems of ambiguity, and cite Baker's draft [Bak11] as a solution.

A few months later, Boutier and al. [BC13], proposed using the disambiguation algorithm presented in this paper, but without defining it formally. They also noticed the interoperability issues.

The first automated implementation of source-sensitive routing known to us was done by Markus Stenberg[2], and works by injecting source-specific default routes determined from IPv6 router advertisements directly into the RIB, independently from the routing protocol. We believe our implementation to be the first full implementation of the source-sensitive scheme.

8. 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 implent source-sensitive routing, and obtained experimental results that prove that source-sensitive routing can be usefully exploited by the transport layer protocol MPTCP.

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-specific 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 planning to modify the *Mosh* [WB12] remote shell replacement to make use of multiple local addresses.

9. SOFTWARE AVAILABILITY

The source-sensitive variant of the `babeld` implementation of the Babel routing protocol is available from *to be added in the final version*.

10. REFERENCES

[AGKT98] George Apostolopoulos, Roch Guérin, Sanjay Kamat, and Satish K. Tripathi. Quality of service based routing: A performance perspective. *SIGCOMM Comput. Commun. Rev.*, 28(4):17–28, October 1998.

[ASNN07] J. Abley, P. Savola, and G. Neville-Neil. Deprecation of Type 0 Routing Headers in IPv6. RFC 5095 (Proposed Standard), December 2007.

[Bak11] F. Baker. Routing in a Traffic Class. Internet-Draft draft-baker-fun-routing-class-00, IETF Secretariat, July 2011.

[Bak13] F. Baker. IPv6 Source/Destination Routing using OSPFv3. Internet-Draft draft-baker-ipv6-ospf-dst-src-routing-03, IETF Secretariat, August 2013.

[BC13] M. Boutier and J. Chroboczek. Source-specific Routing. Internet-Draft draft-boutier-homenet-source-specific-routing-00, IETF Secretariat, July 2013.

[BS04] F. Baker and P. Savola. Ingress Filtering for Multihomed Networks. RFC 3704 (Best

[2] `https://github.com/fingon/hnet-core`

Current Practice), March 2004.

[Chr11] J. Chroboczek. The Babel Routing Protocol. RFC 6126 (Experimental), April 2011.

[FS00] P. Ferguson and D. Senie. Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing. RFC 2827 (Best Current Practice), May 2000. Updated by RFC 3704.

[LHH95] W.C. Lee, M.G. Hluchyi, and P.A. Humblet. Routing subject to quality of service constraints in integrated communication networks. *Network, IEEE*, 9(4):46–55, 1995.

[RPB+12] 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.

[SRC80] Jerome H Saltzer, David P Reed, and David D Clark. Source routing for campus-wide internet transport. In *Proc. IFIP WG 6.4 Int'l Workshop on Local Networks*, pages 1–23, 1980.

[TC13] O. Troan and L. Colitti. IPv6 Multihoming with Source Address Dependent Routing (SADR). Internet-Draft draft-troan-homenet-sadr-01, IETF Secretariat, September 2013.

[WB12] Keith Winstein and Hari Balakrishnan. Mosh: An Interactive Remote Shell for Mobile Clients. In *USENIX Annual Technical Conference*, Boston, MA, June 2012.

[XYWW13] M. Xu, S. Yang, J. Wu, and D. Wang. Two Dimensional-IP Routing Protocol in Home Networks. Internet-Draft draft-xu-homenet-twod-ip-routing-01, IETF Secretariat, August 2013.