Network Working Group
Internet-Draft
Obsoletes: 6126,7557 (if approved)
Intended status: Standards Track
Expires: May 2, 2018

J. Chroboczek
IRIF, University of Paris-Diderot
D. Schinazi
Apple Inc.

October 29, 2017

The Babel Routing Protocol
draft-ietf-babel-rfc6126bis-04

Abstract

Babel is a loop-avoiding distance-vector routing protocol that is robust and efficient both in ordinary wired networks and in wireless mesh networks.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

Babel is a loop-avoiding distance-vector routing protocol that is designed to be robust and efficient both in networks using prefix-based routing and in networks using flat routing ("mesh networks"), and both in relatively stable wired networks and in highly dynamic wireless networks.

1.1. Features

The main property that makes Babel suitable for unstable networks is that, unlike naive distance-vector routing protocols [RIP], it strongly limits the frequency and duration of routing pathologies such as routing loops and black-holes during reconvergence. Even after a mobility event is detected, a Babel network usually remains loop-free. Babel then quickly reconverges to a configuration that preserves the loop-freedom and connectedness of the network, but is not necessarily optimal; in many cases, this operation requires no packet exchanges at all. Babel then slowly converges, in a time on the scale of minutes, to an optimal configuration. This is achieved by using sequenced routes, a technique pioneered by Destination-Sequenced Distance-Vector routing [DSDV].

More precisely, Babel has the following properties:

- when every prefix is originated by at most one router, Babel never suffers from routing loops;

- when a single prefix is originated by multiple routers, Babel may occasionally create a transient routing loop for this particular prefix; this loop disappears in a time proportional to its diameter, and never again (up to an arbitrary garbage-collection (GC) time) will the routers involved participate in a routing loop for the same prefix;

- assuming bounded packet loss rates, any routing black-holes that may appear after a mobility event are corrected in a time at most proportional to the network's diameter.

Babel has provisions for link quality estimation and for fairly arbitrary metrics. When configured suitably, Babel can implement shortest-path routing, or it may use a metric based, for example, on measured packet loss.

Babel nodes will successfully establish an association even when they are configured with different parameters. For example, a mobile node that is low on battery may choose to use larger time constants (hello and update intervals, etc.) than a node that has access to wall power. Conversely, a node that detects high levels of mobility may choose to use smaller time constants. The ability to build such heterogeneous networks makes Babel particularly adapted to the unmanaged and wireless environment.

Finally, Babel is a hybrid routing protocol, in the sense that it can carry routes for multiple network-layer protocols (IPv4 and IPv6), whichever protocol the Babel packets are themselves being carried over.

1.2. Limitations

Babel has two limitations that make it unsuitable for use in some environments. First, Babel relies on periodic routing table updates rather than using a reliable transport; hence, in large, stable networks it generates more traffic than protocols that only send updates when the network topology changes. In such networks, protocols such as OSPF [OSPF], IS-IS [IS-IS], or the Enhanced Interior Gateway Routing Protocol (EIGRP) [EIGRP] might be more suitable.

Second, unless the optional algorithm described in Section 3.5.5 is implemented, Babel does impose a hold time when a prefix is retracted. While this hold time does not apply to the exact prefix being retracted, and hence does not prevent fast reconvergence should it become available again, it does apply to any shorter prefix that covers it. This may make those implementations of Babel that do not implement the optional algorithm described in Section 3.5.5 unsuitable for use in networks that implement automatic prefix aggregation.

1.3. Specification of Requirements

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

2. Conceptual Description of the Protocol

Babel is a loop-avoiding distance vector protocol: it is based on the Bellman-Ford protocol, just like the venerable RIP [RIP], but includes a number of refinements that either prevent loop formation altogether, or ensure that a loop disappears in a timely manner and doesn’t form again.

Conceptually, Bellman-Ford is executed in parallel for every source of routing information (destination of data traffic). In the following discussion, we fix a source S; the reader will recall that the same algorithm is executed for all sources.

2.1. Costs, Metrics and Neighbourship

As many routing algorithms, Babel computes costs of links between any two neighbouring nodes, abstract values attached to the edges between two nodes. We write C(A, B) for the cost of the edge from node A to node B.

Given a route between any two nodes, the metric of the route is the sum of the costs of all the edges along the route. The goal of the routing algorithm is to compute, for every source S, the tree of routes of lowest metric to S.

Costs and metrics need not be integers. In general, they can be values in any algebra that satisfies two fairly general conditions (Section 3.5.2).

A Babel node periodically sends Hello messages to all of its neighbours; it also periodically sends an IHU ("I Heard You") message to every neighbour from which it has recently heard a Hello. From the information derived from Hello and IHU messages received from its neighbour B, a node A computes the cost C(A, B) of the link from A to B.

2.2. The Bellman-Ford Algorithm

Every node A maintains two pieces of data: its estimated distance to S, written D(A), and its next-hop router to S, written NH(A). Initially, D(S) = 0, D(A) is infinite, and NH(A) is undefined.

Periodically, every node B sends to all of its neighbours a route update, a message containing D(B). When a neighbour A of B receives the route update, it checks whether B is its selected next hop; if that is the case, then NH(A) is set to B, and D(A) is set to C(A, B) + D(B). If that is not the case, then A compares C(A, B) + D(B) to its current value of D(A). If that value is smaller, meaning that the received update advertises a route that is better than the currently selected route, then NH(A) is set to B, and D(A) is set to C(A, B) + D(B).

A number of refinements to this algorithm are possible, and are used by Babel. In particular, convergence speed may be increased by sending unscheduled "triggered updates" whenever a major change in the topology is detected, in addition to the regular, scheduled updates. Additionally, a node may maintain a number of alternate routes, which are being advertised by neighbours other than its selected neighbour, and which can be used immediately if the selected route were to fail.

2.3. Transient Loops in Bellman-Ford

It is well known that a naive application of Bellman-Ford to distributed routing can cause transient loops after a topology change. Consider for example the following topology:

```
         B
      1 /|
   1   / |
S --- A  |1
       \ |
      1 \|
         C
```

After convergence, D(B) = D(C) = 2, with NH(B) = NH(C) = A.

Suppose now that the link between S and A fails:

```
         B
      1 /|
       / |
S     A  |1
       \ |
      1 \|
         C
```

When it detects the failure of the link, A switches its next hop to B (which is still advertising a route to S with metric 2), and advertises a metric equal to 3, and then advertises a new route with metric 3. This process of nodes changing selected neighbours and increasing their metric continues until the advertised metric reaches "infinity", a value larger than all the metrics that the routing protocol is able to carry.

2.4. Feasibility Conditions

Bellman-Ford is a very robust algorithm: its convergence properties are preserved when routers delay route acquisition or when they discard some updates. Babel routers discard received route announcements unless they can prove that accepting them cannot possibly cause a routing loop.

More formally, we define a condition over route announcements, known as the "feasibility condition", that guarantees the absence of routing loops whenever all routers ignore route updates that do not satisfy the feasibility condition. In effect, this makes Bellman-Ford into a family of routing algorithms, parameterised by the feasibility condition.

Many different feasibility conditions are possible. For example, BGP can be modelled as being a distance-vector protocol with a (rather drastic) feasibility condition: a routing update is only accepted when the receiving node's AS number is not included in the update's AS-Path attribute (note that BGP's feasibility condition does not ensure the absence of transient "micro-loops" during reconvergence).

Another simple feasibility condition, used in the Destination-Sequenced Distance-Vector (DSDV) routing protocol [DSDV] and in the Ad hoc On-Demand Distance Vector (AODV) protocol, stems from the following observation: a routing loop can only arise after a router has switched to a route with a larger metric than the route that it had previously selected. Hence, one could decide that a route is feasible only when its metric at the local node would be no larger than the metric of the currently selected route, i.e., an announcement carrying a metric D(B) is accepted by A when C(A, B) + D(B) <= D(A). If all routers obey this constraint, then the metric at every router is nonincreasing, and the following invariant is always preserved: if A has selected B as its successor, then D(B) < D(A), which implies that the forwarding graph is loop-free.

Babel uses a slightly more refined feasibility condition, derived from EIGRP [DUAL]. Given a router A, define the feasibility distance of A, written FD(A), as the smallest metric that A has ever advertised for S to any of its neighbours. An update sent by a neighbour B of A is feasible when the metric D(B) advertised by B is strictly smaller than A's feasibility distance, i.e., when D(B) < FD(A).

It is easy to see that this latter condition is no more restrictive than DSDV-feasibility. Suppose that node A obeys DSDV-feasibility; then D(A) is nonincreasing, hence at all times D(A) <= FD(A). Suppose now that A receives a DSDV-feasible update that advertises a metric D(B). Since the update is DSDV-feasible, C(A, B) + D(B) <= D(A), hence D(B) < D(A), and since D(A) <= FD(A), D(B) < FD(A).

To see that it is strictly less restrictive, consider the following diagram, where A has selected the route through B, and D(A) = FD(A) = 2. Since D(C) = 1 < FD(A), the alternate route through C is feasible for A, although its metric C(A, C) + D(C) = 5 is larger than that of the currently selected route:

```
   B
1 / \ 1
 /   \
S     A
 \   /
1 \ / 4
   C
```

To show that this feasibility condition still guarantees loop-freedom, recall that at the time when A accepts an update from B, the metric D(B) announced by B is no smaller than FD(B); since it is smaller than FD(A), at that point in time FD(B) < FD(A). Since this property is preserved when A sends updates, it remains true at all times, which ensures that the forwarding graph has no loops.

2.5. Solving Starvation: Sequencing Routes

Obviously, the feasibility conditions defined above cause starvation when a router runs out of feasible routes. Consider the following diagram, where both A and B have selected the direct route to S:

```
   A
1 /|        D(A) = 1
 / |       FD(A) = 1
S  |1
 \ |        D(B) = 2
2 \|       FD(B) = 2
   B
```

Suppose now that the link between A and S breaks:

```
   A
   |
   |       FD(A) = 1
S  |1
 \ |        D(B) = 2
2 \|       FD(B) = 2
   B
```

The only route available from A to S, the one that goes through B, is not feasible: A suffers from spurious starvation. At this point, the whole network must be rebooted in order to solve the starvation; this is essentially what EIGRP does when it performs a global synchronisation of all the routers in the network with the source (the "active" phase of EIGRP).

Babel reacts to starvation in a less drastic manner, by using sequenced routes, a technique introduced by DSDV and adopted by AODV. In addition to a metric, every route carries a sequence number, a nondecreasing integer that is propagated unchanged through the network and is only ever incremented by the source; a pair (s, m), where s is a sequence number and m a metric, is called a distance.

A received update is feasible when either it is more recent than the feasibility distance maintained by the receiving node, or it is equally recent and the metric is strictly smaller. More formally, if FD(A) = (s, m), then an update carrying the distance (s’, m’) is feasible when either s’ > s, or s = s’ and m’ < m.

Assuming the sequence number of S is 137, the diagram above becomes:

```
   A
   |
   |       FD(A) = (137, 1)
S  |1
 \ |        D(B) = (137, 2)
2 \|       FD(B) = (137, 2)
   B
```

After S increases its sequence number, and the new sequence number is propagated to B, we have:

```
   A
   |
   |       FD(A) = (137, 1)
S  |1
 \ |        D(B) = (138, 2)
2 \|       FD(B) = (138, 2)
   B
```

at which point the route through B becomes feasible again.

Note that while sequence numbers are used for determining feasibility, they are not necessarily used in route selection: a node will normally ignore the sequence number when selecting the best route to a given destination (Section 3.6).

2.6. Requests

In DSDV, the sequence number of a source is increased periodically. A route becomes feasible again after the source increases its sequence number, and the new sequence number is propagated through the network, which may, in general, require a significant amount of time.

Babel takes a different approach. When a node detects that it is suffering from a potentially spurious starvation, it sends an explicit request to the source for a new sequence number. This request is forwarded hop by hop to the source, with no regard to the feasibility condition. Upon receiving the request, the source increases its sequence number and broadcasts an update, which is forwarded to the requesting node.

Note that after a change in network topology not all such requests will, in general, reach the source, as some will be sent over links that are now broken. However, if the network is still connected, then at least one among the nodes suffering from spurious starvation has an (unfeasible) route to the source; hence, in the absence of packet loss, at least one such request will reach the source. (Resending requests a small number of times compensates for packet loss.)

Since requests are forwarded with no regard to the feasibility condition, they may, in general, be caught in a forwarding loop; this is avoided by having nodes perform duplicate detection for the requests that they forward.

2.7. Multiple Routers

The above discussion assumes that every prefix is originated by a single router. In real networks, however, it is often necessary to have a single prefix originated by multiple routers: for example, the default route will be originated by all of the edge routers of a routing domain.

Since synchronising sequence numbers between distinct routers is problematic, Babel treats routes for the same prefix as distinct entities when they are originated by different routers: every route announcement carries the router-id of its originating router, and feasibility distances are not maintained per prefix, but per source, where a source is a pair of a router-id and a prefix. In effect, Babel guarantees loop-freedom for the forwarding graph to every source; since the union of multiple acyclic graphs is not in general acyclic, Babel does not in general guarantee loop-freedom when a prefix is originated by multiple routers, but any loops will be broken in a time at most proportional to the diameter of the loop -- as soon as an update has "gone around" the routing loop.

Consider for example the following topology, where A has selected the default route through S, and B has selected the one through S’:

```
           1     1     1
::/0 -- S --- A --- B --- S’ -- ::/0
```

Suppose that both default routes fail at the same time; then nothing prevents A from switching to B, and B simultaneously switching to A. However, as soon as A has successfully advertised the new route to B, the route through A will become unfeasible for B. Conversely, as soon as B will have advertised the route through A, the route through B will become unfeasible for A.

In effect, the routing loop disappears at the latest when routing information has gone around the loop. Since this process can be delayed by lost packets, Babel makes certain efforts to ensure that updates are sent reliably after a router-id change Section 3.7.2.

Additionally, after the routers have advertised the two routes, both sources will be in their source tables, which will prevent them from ever again participating in a routing loop involving routes from S and S’ (up to the source GC time, which, available memory permitting, can be set to arbitrarily large values).

2.8. Overlapping Prefixes

In the above discussion, we have assumed that all prefixes are disjoint, as is the case in flat ("mesh") routing. In practice, however, prefixes may overlap: for example, the default route overlaps with all of the routes present in the network.

After a route fails, it is not correct in general to switch to a route that subsumes the failed route. Consider for example the following configuration:

```
           1     1
::/0 -- A --- B --- C
```

Suppose that node C fails. If B forwards packets destined to C by following the default route, a routing loop will form, and persist until A learns of B’s retraction of the direct route to C. B avoids this pitfall by installing an "unreachable" route after a route is retracted; this route is maintained until it can be guaranteed that the former route has been retracted by all of B’s neighbours (Section 3.5.5).

3. Protocol Operation

Every Babel speaker is assigned a router-id, which is an arbitrary string of 8 octets that is assumed unique across the routing domain. For example, routers-ids could be assigned randomly, or they could derived from a link-layer address. (The protocol encoding is slightly more compact when router-ids are assigned in the same manner as the IPv6 layer assigns host IDs.)

3.1. Message Transmission and Reception

Babel protocol packets are sent in the body of a UDP datagram. Each Babel packet consists of zero or more TLVs. Most TLVs may contain sub-TLVs.

The source address of a Babel packet is always a unicast address, link-local in the case of IPv6. Babel packets may be sent to a well-known (link-local) multicast address or to a (link-local) unicast address. In normal operation, a Babel speaker sends both multicast and unicast packets to its neighbours.

With the exception of Hello TLVs and acknowledgments, all Babel TLVs can be sent to either unicast or multicast addresses, and their semantics does not depend on whether the destination is a unicast or a multicast address. Hence, a Babel speaker does not need to determine the destination address of a packet that it receives in order to interpret it.

A moderate amount of jitter may be applied to packets sent by a Babel speaker: outgoing TLVs are buffered and SHOULD be sent with a small random delay. This is done for two purposes: it avoids synchronisation of multiple Babel speakers across a network [JITTER], and it allows for the aggregation of multiple TLVs into a single packet.

The exact delay and amount of jitter applied to a packet depends on whether it contains any urgent TLVs. Acknowledgment TLVs MUST be sent before the deadline specified in the corresponding request. The particular class of updates specified in Section 3.7.2 MUST be sent in a timely manner. The particular class of request and update TLVs specified in Section 3.8.2 SHOULD be sent in a timely manner.

3.2. Data Structures

In this section, we give a description of the data structures that every Babel speaker maintains. This description is conceptual: a Babel speaker may use different data structures as long as the resulting protocol is the same as the one described in this document.

For example, rather than maintaining a single table containing both selected and unselected (fallback) routes, as described in Section 3.2.6 belong, an actual implementation would probably use two tables, one with selected routes and one with fallback routes.

3.2.1. Sequence number arithmetic

Sequence numbers (seqnos) appear in a number of Babel data structures, and they are interpreted as integers modulo 2^16. For the purposes of this document, arithmetic on sequence numbers is defined as follows.

Given a seqno s and an integer n, the sum of s and n is defined by

```
s + n (modulo 2^16) = (s + n) MOD 2^16
```

or, equivalently,

```
s + n (modulo 2^16) = (s + n) AND 65535
```

where MOD is the modulo operation yielding a non-negative integer and AND is the bitwise conjunction operation.

Given two sequence numbers s and s', the relation s is less than s' (s < s') is defined by

```
s < s' (modulo 2^16) when 0 < ((s' - s) MOD 2^16) < 32768
```

or equivalently

```
s < s' (modulo 2^16) when s /= s' and ((s' - s) AND 32768) = 0.
```

3.2.2. Node Sequence Number

A node's sequence number is a 16-bit integer that is included in route updates sent for routes originated by this node.

A node increments its sequence number (modulo 2^16) whenever it receives a request for a new sequence number (Section 3.8.1.2). A node SHOULD NOT increment its sequence number (seqno) spontaneously, since increasing seqnos makes it less likely that other nodes will have feasible alternate routes when their selected routes fail.

3.2.3. The Interface Table

The interface table contains the list of interfaces on which the node speaks the Babel protocol. Every interface table entry contains the interface's outgoing Multicast Hello seqno, a 16-bit integer that is sent with each Multicast Hello TLV on this interface and is incremented (modulo 2^16) whenever a Multicast Hello is sent. (Note that an interface's Multicast Hello seqno is unrelated to the node's seqno.)

There are two timers associated with each interface table entry -- the multicast hello timer, which governs the sending of scheduled Multicast Hello and IHU packets, and the update timer, which governs the sending of periodic route updates.

3.2.4. The Neighbour Table

The neighbour table contains the list of all neighbouring interfaces from which a Babel packet has been recently received. The neighbour table is indexed by pairs of the form (interface, address), and every neighbour table entry contains the following data:

- the local node's interface over which this neighbour is reachable;

- the address of the neighbouring interface;

- a history of recently received Multicast Hello packets from this neighbour; this can, for example, be a sequence of n bits, for some small value n, indicating which of the n hellos most recently sent by this neighbour have been received by the local node;

- a history of recently received Unicast Hello packets from this neighbour;

- the "transmission cost" value from the last IHU packet received from this neighbour, or FFFF hexadecimal (infinity) if the IHU hold timer for this neighbour has expired;

- the neighbour's expected incoming Multicast Hello sequence number, an integer modulo 2^16.

- the neighbour's expected incoming Unicast Hello sequence number, an integer modulo 2^16.

- the neighbour's outgoing Unicast Hello sequence number, an integer modulo 2^16 that is sent with each Unicast Hello TLV to this neighbour and is incremented (modulo 2^16) whenever a Unicast Hello is sent. (Note that a neighbour's outgoing Unicast Hello seqno is distinct from the interface's outgoing Multicast Hello seqno.)

There are three timers associated with each neighbour entry -- the multicast hello timer, which is initialised from the interval value carried by scheduled Multicast Hello TLVs, the unicast hello timer, which is initialised from the interval value carried by scheduled Unicast Hello TLVs, and the IHU timer, which is initialised to a small multiple of the interval carried in IHU TLVs.

Note that the neighbour table is indexed by IP addresses, not by router-ids: neighbourship is a relationship between interfaces, not between nodes. Therefore, two nodes with multiple interfaces can participate in multiple neighbourship relationships, a situation that can notably arise when wireless nodes with multiple radios are involved.

3.2.5. The Source Table

The source table is used to record feasibility distances. It is indexed by triples of the form (prefix, plen, router-id), and every source table entry contains the following data:

- the prefix (prefix, plen), where plen is the prefix length, that this entry applies to;
- the router-id of a router originating this prefix;
- a pair (seqno, metric), this source's feasibility distance.

There is one timer associated with each entry in the source table -- the source garbage-collection timer. It is initialised to a time on the order of minutes and reset as specified in Section 3.7.3.

3.2.6. The Route Table

The route table contains the routes known to this node. It is indexed by triples of the form (prefix, plen, neighbour), and every route table entry contains the following data:

- the source (prefix, plen, router-id) for which this route is advertised;
- the neighbour that advertised this route;
- the metric with which this route was advertised by the neighbour, or FFFF hexadecimal (infinity) for a recently retracted route;
- the sequence number with which this route was advertised;
- the next-hop address of this route;

- a boolean flag indicating whether this route is selected, i.e., whether it is currently being used for forwarding and is being advertised.

There is one timer associated with each route table entry -- the route expiry timer. It is initialised and reset as specified in Section 3.5.4.

Note that there are two distinct (seqno, metric) pairs associated to each route: the route’s distance, which is stored in the route table, and the feasibility distance, stored in the source table and shared between all routes with the same source.

3.2.7. The Table of Pending Seqno Requests

The table of pending seqno requests contains a list of seqno requests that the local node has sent (either because they have been originated locally, or because they were forwarded) and to which no reply has been received yet. This table is indexed by triples of the form (prefix, plen, router-id), and every entry in this table contains the following data:

- the prefix, router-id, and seqno being requested;

- the neighbour, if any, on behalf of which we are forwarding this request;

- a small integer indicating the number of times that this request will be resent if it remains unsatisfied.

There is one timer associated with each pending seqno request; it governs both the resending of requests and their expiry.

3.3. Acknowledgments and acknowledgment requests

A Babel speaker may request that a neighbour receiving a given packet reply with an explicit acknowledgment within a given time. While the use of acknowledgment requests is optional, every Babel speaker MUST be able to reply to such a request.

An acknowledgment MUST be sent to a unicast destination. On the other hand, acknowledgment requests may be sent to either unicast or multicast destinations, in which case they request an acknowledgment from all of the receiving nodes.

When to request acknowledgments is a matter of local policy; the simplest strategy is to never request acknowledgments and to rely on periodic updates to ensure that any reachable routes are eventually propagated throughout the routing domain. In order to improve convergence speed and reduce the amount of control traffic, acknowledgment requests MAY be used in order to reliably send urgent updates (Section 3.7.2) and retractions (Section 3.5.5), especially when the number of neighbours on a given interface is small. Since Babel is designed to deal gracefully with packet loss on unreliable media, sending all packets with acknowledgment requests is not necessary, and NOT RECOMMENDED, as the acknowledgments cause additional traffic and may force additional Address Resolution Protocol (ARP) or Neighbour Discovery (ND) exchanges.

3.4. Neighbour Acquisition

Neighbour acquisition is the process by which a Babel node discovers the set of neighbours heard over each of its interfaces and ascertains bidirectional reachability. On unreliable media, neighbour acquisition additionally provides some statistics that may be useful for link quality computation.

Before it can exchange routing information with a neighbour, a Babel node MUST create an entry for that neighbour in the neighbour table. When to do that is implementation-specific; suitable strategies include creating an entry when any Babel packet is received, or creating an entry when a Hello TLV is parsed. Similarly, in order to conserve system resources, an implementation SHOULD discard an entry when it has been unused for long enough; suitable strategies include dropping the neighbour after a timeout, and dropping a neighbour when the associated Hello histories become empty (see Appendix A.2).

3.4.1. Reverse Reachability Detection

Every Babel node sends Hello TLVs to its neighbours to indicate that it is alive, at regular or irregular intervals. Each Hello TLV carries an increasing (modulo 2^16) sequence number and an upper bound on the time interval until the next Hello of the same type (see below). If the time interval is set to 0, then the Hello TLV does not establish a new promise: the deadline carried by the previous Hello of the same type still applies to the next Hello (if the most recent scheduled Hello of the right kind was received at time t0 and carried interval i, then the previous promise of sending another Hello before time t0 + i still holds). We say that a Hello is "scheduled" if it carries a non-zero interval, and "unscheduled" otherwise.

There are two kinds of Hellos: Multicast Hellos, which use a per-interface Hello counter, and Unicast Hellos, which use a per-neighbour counter. A Multicast Hello with a given seqno MUST be sent to all neighbours on a given interface, either by sending it to a multicast address or by sending it to one unicast address per neighbour (hence, the term "Multicast Hello" is a slight misnomer). A Unicast Hello carrying a given seqno should normally be sent to just one neighbour (over unicast), since the sequence numbers of different neighbours are not in general synchronised.

Multicast Hellos sent over multicast can be used for neighbour discovery; hence, a node SHOULD send periodic (scheduled) Multicast Hellos unless neighbour discovery is performed by means outside of the Babel protocol. A node MAY send Unicast Hellos or unscheduled Hellos of either kind for any reason, such as reducing the amount of multicast traffic or improving reliability on link technologies with poor support for link-layer multicast.

A node MAY send a scheduled Hello ahead of time. A node MAY change its scheduled Hello interval. The Hello interval MAY be decreased at any time; it MAY be increased immediately before sending a Hello TLV, but SHOULD NOT be increased at other times. (Equivalently, a node SHOULD send a scheduled Hello immediately after increasing its Hello interval.)

How to deal with received Hello TLVs and what statistics to maintain are considered local implementation matters; typically, a node will maintain some sort of history of recently received Hellos. An example of a suitable algorithm is described in Appendix A.1.

After receiving a Hello, or determining that it has missed one, the node recomputes the association’s cost (Section 3.4.3) and runs the route selection procedure (Section 3.6).

3.4.2. Bidirectional Reachability Detection

In order to establish bidirectional reachability, every node sends periodic IHU ("I Heard You") TLVs to each of its neighbours. Since IHUs carry an explicit interval value, they MAY be sent less often than Hellos in order to reduce the amount of routing traffic in dense networks; in particular, they SHOULD be sent less often than Hellos over links with little packet loss. While IHUs are conceptually unicast, they MAY be sent to a multicast address in order to avoid an ARP or Neighbour Discovery exchange and to aggregate multiple IHUs into a single packet.

In addition to the periodic IHUs, a node MAY, at any time, send an unscheduled IHU packet. It MAY also, at any time, decrease its IHU interval, and it MAY increase its IHU interval immediately before sending an IHU, but SHOULD NOT increase it at any other time. (Equivalently, a node SHOULD send an extra IHU immediately after increasing its Hello interval.)

Every IHU TLV contains two pieces of data: the link's rxcost (reception cost) from the sender's perspective, used by the neighbour for computing link costs (Section 3.4.3), and the interval between periodic IHU packets. A node receiving an IHU sets the value of the txcost (transmission cost) maintained in the neighbour table to the value contained in the IHU, and resets the IHU timer associated to this neighbour to a small multiple of the interval value received in the IHU. When a neighbour's IHU timer expires, the neighbour's txcost is set to infinity.

After updating a neighbour's txcost, the receiving node recomputes the neighbour's cost (Section 3.4.3) and runs the route selection procedure (Section 3.6).

3.4.3. Cost Computation

A neighbourship association's link cost is computed from the values maintained in the neighbour table: the statistics kept in the neighbour table about the reception of Hellos, and the txcost computed from received IHU packets.

For every neighbour, a Babel node computes a value known as this neighbour's rxcost. This value is usually derived from the Hello history, which may be combined with other data, such as statistics maintained by the link layer. The rxcost is sent to a neighbour in each IHU.

Since nodes do not necessarily send periodic Unicast Hellos but do usually send periodic Multicast Hellos (Section 3.4.1), a node SHOULD use an algorithm that yields a finite rxcost when only Multicast Hellos are received, unless interoperability with nodes that only send Multicast Hellos is not required.

How the txcost and rxcost are combined in order to compute a link's cost is a matter of local policy; as far as Babel's correctness is concerned, only the following conditions MUST be satisfied:

- the cost is strictly positive;

- if no Hello TLVs of either kind were received recently, then the cost is infinite;

- if the txcost is infinite, then the cost is infinite.

Note that while this document does not constrain cost computation any further, not all cost computation strategies will give good results. See Appendix A.2 for examples of strategies for computing a link's cost that are known to work well in practice.

3.5. Routing Table Maintenance

Conceptually, a Babel update is a quintuple (prefix, plen, router-id, seqno, metric), where (prefix, plen) is the prefix for which a route is being advertised, router-id is the router-id of the router originating this update, seqno is a nondecreasing (modulo 2^16) integer that carries the originating router seqno, and metric is the announced metric.

Before being accepted, an update is checked against the feasibility condition (Section 3.5.1), which ensures that the route does not create a routing loop. If the feasibility condition is not satisfied, the update is either ignored or prevents the route from being selected, as described in Section 3.5.4. If the feasibility condition is satisfied, then the update cannot possibly cause a routing loop.

3.5.1. The Feasibility Condition

The feasibility condition is applied to all received updates. The feasibility condition compares the metric in the received update with the metrics of the updates previously sent by the receiving node; updates that fail the feasibility condition, and therefore have metrics large enough to cause a routing loop, are either ignored or prevent the resulting route from being selected.

A feasibility distance is a pair (seqno, metric), where seqno is an integer modulo 2^16 and metric is a positive integer. Feasibility distances are compared lexicographically, with the first component inverted: we say that a distance (seqno, metric) is strictly better than a distance (seqno’, metric’), written

```
   (seqno, metric) < (seqno’, metric’)
```

when

```
   seqno > seqno’ or (seqno = seqno’ and metric < metric’)
```

where sequence numbers are compared modulo 2^16.

Given a source (prefix, plen, router-id), a node’s feasibility distance for this source is the minimum, according to the ordering defined above, of the distances of all the finite updates ever sent by this particular node for the prefix (prefix, plen) and the given router-id. Feasibility distances are maintained in the source table, the exact procedure is given in Section 3.7.3.

A received update is feasible when either it is a retraction (its metric is FFFF hexadecimal), or the advertised distance is strictly better, in the sense defined above, than the feasibility distance for the corresponding source. More precisely, a route advertisement carrying the quintuple (prefix, plen, router-id, seqno, metric) is feasible if one of the following conditions holds:

- metric is infinite; or
- no entry exists in the source table indexed by (prefix, plen, router-id); or
- an entry (prefix, plen, router-id, seqno’, metric’) exists in the source table, and either
    - seqno’ < seqno or
    - seqno = seqno’ and metric < metric’.

Note that the feasibility condition considers the metric advertised by the neighbour, not the route’s metric; hence, a fluctuation in a neighbour’s cost cannot render a selected route unfeasible. Note further that retractions (updates with infinite metric) are always feasible, since they cannot possibly cause a routing loop.

3.5.2. Metric Computation

A route’s metric is computed from the metric advertised by the neighbour and the neighbour’s link cost. Just like cost computation, metric computation is considered a local policy matter; as far as Babel is concerned, the function M(c, m) used for computing a metric from a locally computed link cost and the metric advertised by a neighbour MUST only satisfy the following conditions:

- if c is infinite, then M(c, m) is infinite;
- M is strictly monotonic: M(c, m) > m.

Additionally, the metric SHOULD satisfy the following condition:

- M is isotonic: if m <= m’, then M(c, m) <= M(c, m’).

Note that while strict monotonicity is essential to the integrity of the network (persistent routing loops may arise if it is not satisfied), isotonicity is not: if it is not satisfied, Babel will still converge to a loop-free configuration, but might not reach a global optimum (in fact, a global optimum may not even exist).

As with cost computation, not all strategies for computing route metrics will give good results. In particular, some metrics are more likely than others to lead to routing instabilities (route flapping). In Appendix A.3, we give a number of examples of strictly monotonic, isotonic routing metrics that are known to work well in practice.

3.5.3. Encoding of Updates

In a large network, the bulk of Babel traffic consists of route updates; hence, some care has been given to encoding them efficiently. An Update TLV itself only contains the prefix, seqno, and metric, while the next hop is derived either from the network-layer source address of the packet or from an explicit Next Hop TLV in the same packet. The router-id is derived from a separate Router-Id TLV in the same packet, which optimises the case when multiple updates are sent with the same router-id.

Additionally, a prefix of the advertised prefix can be omitted in an Update TLV, in which case it is copied from a previous Update TLV in the same packet -- this is known as address compression (Section 4.6.9).

Finally, as a special optimisation for the case when a router-id coincides with the interface-id part of an IPv6 address, the router-id can optionally be derived from the low-order bits of the advertised prefix.

The encoding of updates is described in detail in Section 4.6.

3.5.4. Route Acquisition

When a Babel node receives an update (prefix, plen, router-id, seqno, metric) from a neighbour neigh with a link cost value equal to cost, it checks whether it already has a route table entry indexed by (prefix, plen, neigh).

If no such entry exists:

- if the update is unfeasible, it MAY be ignored;

- if the metric is infinite (the update is a retraction of a route we do not know about), the update is ignored;

- otherwise, a new entry is created in the route table, indexed by (prefix, plen, neigh), with source equal to (prefix, plen, router-id), seqno equal to seqno and an advertised metric equal to the metric carried by the update.

If such an entry exists:

- if the entry is currently selected, the update is unfeasible, and the router-id of the update is equal to the router-id of the entry, then the update MAY be ignored;

- otherwise, the entry's sequence number, advertised metric, metric, and router-id are updated and, if the advertised metric is not infinite, the route's expiry timer is reset to a small multiple of the Interval value included in the update. If the update is unfeasible, then the (now unfeasible) entry MUST be immediately unselected. If the update caused the router-id of the entry to change, an update (possibly a retraction) MUST be sent in a timely manner (see Section 3.7.2).

Note that the route table may contain unfeasible routes, either because they were created by an unfeasible update or due to a metric fluctuation. Such routes are never selected, since they are not known to be loop-free; should all the feasible routes become unusable, however, the unfeasible routes can be made feasible and therefore possible to select by sending requests along them (see Section 3.8.2).

When a route's expiry timer triggers, the behaviour depends on whether the route's metric is finite. If the metric is finite, it is set to infinity and the expiry timer is reset. If the metric is already infinite, the route is flushed from the route table.

After the route table is updated, the route selection procedure (Section 3.6) is run.

3.5.5. Hold Time

When a prefix P is retracted, because all routes are unfeasible or have an infinite metric (whether due to the expiry timer or to other reasons), and a shorter prefix P' that covers P is reachable, P' cannot in general be used for routing packets destined to P without running the risk of creating a routing loop (Section 2.8).

To avoid this issue, whenever a prefix P is retracted, a route table entry with infinite metric is maintained as described in Section 3.5.4 above. As long as this entry is maintained, packets destined to an address within P MUST NOT be forwarded by following a route for a shorter prefix. This entry is removed as soon as a finite-metric update for prefix P is received and the resulting route selected. If no such update is forthcoming, the infinite metric entry SHOULD be maintained at least until it is guaranteed that no neighbour has selected the current node as next-hop for prefix P. This can be achieved by either:

- waiting until the route's expiry timer has expired (Section 3.5.4);

- sending a retraction with an acknowledgment request (Section 3.3) to every reachable neighbour that has not explicitly retracted prefix P and waiting for all acknowledgments.

The former option is simpler and ensures that at that point, any routes for prefix P pointing at the current node have expired. However, since the expiry time can be as high as a few minutes, doing that prevents automatic aggregation by creating spurious black-holes for aggregated routes. The latter option is RECOMMENDED as it dramatically reduces the time for which a prefix is unreachable in the presence of aggregated routes.

3.6. Route Selection

Route selection is the process by which a single route for a given prefix is selected to be used for forwarding packets and to be re-advertised to a node's neighbours.

Babel is designed to allow flexible route selection policies. As far as the protocol's correctness is concerned, the route selection policy MUST only satisfy the following properties:

- a route with infinite metric (a retracted route) is never selected;

- an unfeasible route is never selected.

Note, however, that Babel does not naturally guarantee the stability of routing, and configuring conflicting route selection policies on different routers may lead to persistent route oscillation.

Route selection is a difficult problem, since a good route selection policy needs to take into account multiple mutually contradictory criteria; in roughly decreasing order of importance, these are:

- routes with a small metric should be preferred to routes with a large metric;

- switching router-ids should be avoided;

- routes through stable neighbours should be preferred to routes through unstable ones;

- stable routes should be preferred to unstable ones;
- switching next hops should be avoided.

A simple but useful strategy is to choose the feasible route with the smallest metric, with a small amount of hysteresis applied to avoid switching router-ids too often.

After the route selection procedure is run, triggered updates (Section 3.7.2) and requests (Section 3.8.2) are sent.

3.7. Sending Updates

A Babel speaker advertises to its neighbours its set of selected routes. Normally, this is done by sending one or more multicast packets containing Update TLVs on all of its connected interfaces; however, on link technologies where multicast is significantly more expensive than unicast, a node MAY choose to send multiple copies of updates in unicast packets, especially when the number of neighbours is small.

Additionally, in order to ensure that any black-holes are reliably cleared in a timely manner, a Babel node sends retractions (updates with an infinite metric) for any recently retracted prefixes.

If an update is for a route injected into the Babel domain by the local node (e.g., it carries the address of a local interface, the prefix of a directly attached network, or a prefix redistributed from a different routing protocol), the router-id is set to the local node's router-id, the metric is set to some arbitrary finite value (typically 0), and the seqno is set to the local router's sequence number.

If an update is for a route learned from another Babel speaker, the router-id and sequence number are copied from the route table entry, and the metric is computed as specified in Section 3.5.2.

3.7.1. Periodic Updates

Every Babel speaker periodically advertises all of its selected routes on all of its interfaces, including any recently retracted routes. Since Babel doesn't suffer from routing loops (there is no "counting to infinity") and relies heavily on triggered updates (Section 3.7.2), this full dump only needs to happen infrequently.

3.7.2. Triggered Updates

In addition to periodic routing updates, a Babel speaker sends unscheduled, or triggered, updates in order to inform its neighbours of a significant change in the network topology.

A change of router-id for the selected route to a given prefix may be indicative of a routing loop in formation; hence, a node MUST send a triggered update in a timely manner whenever it changes the selected router-id for a given destination. Additionally, it SHOULD make a reasonable attempt at ensuring that all reachable neighbours receive this update.

There are two strategies for ensuring that. If the number of neighbours is small, then it is reasonable to send the update together with an acknowledgment request; the update is resent until all neighbours have acknowledged the packet, up to some number of times. If the number of neighbours is large, however, requesting acknowledgments from all of them might cause a non-negligible amount of network traffic; in that case, it may be preferable to simply repeat the update some reasonable number of times (say, 5 for wireless and 2 for wired links).

A route retraction is somewhat less worrying: if the route retraction doesn't reach all neighbours, a black-hole might be created, which, unlike a routing loop, does not endanger the integrity of the network. When a route is retracted, a node SHOULD send a triggered update and SHOULD make a reasonable attempt at ensuring that all neighbours receive this retraction.

Finally, a node MAY send a triggered update when the metric for a given prefix changes in a significant manner, due to a received update, because a link's cost has changed, or because a different next hop has been selected. A node SHOULD NOT send triggered updates for other reasons, such as when there is a minor fluctuation in a route's metric, when the selected next hop changes, or to propagate a new sequence number (except to satisfy a request, as specified in Section 3.8).

3.7.3. Maintaining Feasibility Distances

Before sending an update (prefix, plen, router-id, seqno, metric) with finite metric (i.e., not a route retraction), a Babel node updates the feasibility distance maintained in the source table. This is done as follows.

If no entry indexed by (prefix, plen, router-id) exists in the source table, then one is created with value (prefix, plen, router-id, seqno, metric).

If an entry (prefix, plen, router-id, seqno’, metric’) exists, then it is updated as follows:

- if seqno > seqno’, then seqno’ := seqno, metric’ := metric;

- if seqno = seqno’ and metric’ > metric, then metric’ := metric;

- otherwise, nothing needs to be done.

The garbage-collection timer for the entry is then reset. Note that the feasibility distance is not updated and the garbage-collection timer is not reset when a retraction (an update with infinite metric) is sent.

When the garbage-collection timer expires, the entry is removed from the source table.

3.7.4. Split Horizon

When running over a transitive, symmetric link technology, e.g., a point-to-point link or a wired LAN technology such as Ethernet, a Babel node SHOULD use an optimisation known as split horizon. When split horizon is used on a given interface, a routing update for prefix P is not sent on the particular interface over which the selected route towards prefix P was learnt.

Split horizon SHOULD NOT be applied to an interface unless the interface is known to be symmetric and transitive; in particular, split horizon is not applicable to decentralised wireless link technologies (e.g., IEEE 802.11 in ad hoc mode) when routing updates are sent over multicast.

3.8. Explicit Requests

In normal operation, a node’s route table is populated by the regular and triggered updates sent by its neighbours. Under some circumstances, however, a node sends explicit requests in order to cause a resynchronisation with the source after a mobility event or to prevent a route from spuriously expiring.

The Babel protocol provides two kinds of explicit requests: route requests, which simply request an update for a given prefix, and seqno requests, which request an update for a given prefix with a specific sequence number. The former are never forwarded; the latter are forwarded if they cannot be satisfied by the receiver.

3.8.1. Handling Requests

Upon receiving a request, a node either forwards the request or sends an update in reply to the request, as described in the following sections. If this causes an update to be sent, the update is either sent to a multicast address on the interface on which the request was received, or to the unicast address of the neighbour that sent the request.

The exact behaviour is different for route requests and seqno requests.

3.8.1.1. Route Requests

When a node receives a route request for a given prefix, it checks its route table for a selected route to this exact prefix. If such a route exists, it MUST send an update (over unicast or over multicast); if such a route does not, it MUST send a retraction for that prefix.

When a node receives a wildcard route request, it SHOULD send a full route table dump. Full route dumps MAY be rate-limited, especially if they are sent over multicast.

3.8.1.2. Seqno Requests

When a node receives a seqno request for a given router-id and sequence number, it checks whether its route table contains a selected entry for that prefix. If a selected route for the given prefix exists, it has finite metric, and either the router-ids are different or the router-ids are equal and the entry’s sequence number is no smaller (modulo 2^16) than the requested sequence number, the node MUST send an update for the given prefix. If the router-ids match but the requested seqno is larger (modulo 2^16) than the route entry’s, the node compares the router-id against its own router-id. If the router-id is its own, then it increases its sequence number by 1 (modulo 2^16) and sends an update. A node MUST NOT increase its sequence number by more than 1 in response to a seqno request.

Otherwise, if the requested router-id is not its own, the received request’s hop count is 2 or more, and the node is advertising the prefix to its neighbours, the node selects a neighbour to forward the request to as follows:

- if the node has one or more feasible routes toward the requested prefix with a next hop that is not the requesting node, then the node MUST forward the request to the next hop of one such route;

- otherwise, if the node has one or more (not necessarily feasible) routes to the requested prefix with a next hop that is not the requesting node, then the node SHOULD forward the request to the next hop of one such route.

In order to actually forward the request, the node decrements the hop count and sends the request in a unicast packet destined to the selected neighbour.

A node SHOULD maintain a list of recently forwarded seqno requests and forward the reply (an update with a sufficiently large seqno) in a timely manner. A node SHOULD compare every incoming seqno request against its list of recently forwarded seqno requests and avoid forwarding it if it is redundant (i.e., if it has recently sent a request with the same prefix, router-id and a seqno that is not smaller modulo 2^16).

Since the request-forwarding mechanism does not necessarily obey the feasibility condition, it may get caught in routing loops; hence, requests carry a hop count to limit the time during which they remain in the network. However, since requests are only ever forwarded as unicast packets, the initial hop count need not be kept particularly low, and performing an expanding horizon search is not necessary. A single request MUST NOT be duplicated: it MUST NOT be forwarded to a multicast address, and it MUST NOT be forwarded to multiple neighbours. However, if a seqno request is resent by its originator, the subsequent copies MAY be forwarded to a different neighbour than the initial one.

3.8.2. Sending Requests

A Babel node MAY send a route or seqno request at any time, to a multicast or a unicast address; there is only one case when originating requests is required (Section 3.8.2.1).

3.8.2.1. Avoiding Starvation

When a route is retracted or expires, a Babel node usually switches to another feasible route for the same prefix. It may be the case, however, that no such routes are available.

A node that has lost all feasible routes to a given destination but still has unexpired unfeasible routes to that destination MUST send a seqno request; if it doesn't have any such routes, it MAY still send a seqno request. The router-id of the request is set to the router-id of the route that it has just lost, and the requested seqno is the value contained in the source table plus 1.

If the node has any (unfeasible) routes to the requested destination, then it MUST send the request to at least one of the next-hop neighbours that advertised these routes, and SHOULD send it to all of them; in any case, it MAY send the request to any other neighbours, whether they advertise a route to the requested destination or not. A simple implementation strategy is therefore to unconditionally multicast the request over all interfaces.

Similar requests will be sent by other nodes that are affected by the route's loss. If the network is still connected, and assuming no packet loss, then at least one of these requests will be forwarded to the source, resulting in a route being advertised with a new sequence number. (Due to duplicate suppression, only a small number of such requests will actually reach the source.)

In order to compensate for packet loss, a node SHOULD repeat such a request a small number of times if no route becomes feasible within a short time. In the presence of heavy packet loss, however, all such requests might be lost; in that case, the mechanism in the next section will eventually ensure that a new seqno is received.

3.8.2.2. Dealing with Unfeasible Updates

When a route's metric increases, a node might receive an unfeasible update for a route that it has currently selected. As specified in Section 3.5.1, the receiving node will either ignore the update or unselect the route.

In order to keep routes from spuriously expiring because they have become unfeasible, a node SHOULD send a unicast seqno request when it receives an unfeasible update for a route that is currently selected. The requested sequence number is computed from the source table as in Section 3.8.2.1 above.

Additionally, since metric computation does not necessarily coincide with the delay in propagating updates, a node might receive an unfeasible update from a currently unselected neighbour that is preferable to the currently selected route (e.g., because it has a much smaller metric); in that case, the node SHOULD send a unicast seqno request to the neighbour that advertised the preferable update.

3.8.2.3. Preventing Routes from Expiring

In normal operation, a route's expiry timer never triggers: since a route's hold time is computed from an explicit interval included in Update TLVs, a new update (possibly a retraction) should arrive in time to prevent a route from expiring.

In the presence of packet loss, however, it may be the case that no update is successfully received for an extended period of time, causing a route to expire. In order to avoid such spurious expiry, shortly before a selected route expires, a Babel node SHOULD send a unicast route request to the neighbour that advertised this route; since nodes always send either updates or retractions in response to non-wildcard route requests (Section 3.8.1.1), this will usually result in the route being either refreshed or retracted.

3.8.2.4. Acquiring New Neighbours

In order to speed up convergence after a mobility event, a node MAY send a unicast wildcard request after acquiring a new neighbour. Additionally, a node MAY send a small number of multicast wildcard requests shortly after booting. Note however that doing that carelessly can cause serious congestion when a whole network is rebooted, especially on link layers with high per-packet overhead (e.g., IEEE 802.11).

4. Protocol Encoding

A Babel packet is sent as the body of a UDP datagram, with network-layer hop count set to 1, destined to a well-known multicast address or to a unicast address, over IPv4 or IPv6; in the case of IPv6, these addresses are link-local. Both the source and destination UDP port are set to a well-known port number. A Babel packet MUST be silently ignored unless its source address is either a link-local IPv6 address or an IPv4 address belonging to the local network, and its source port is the well-known Babel port. It MAY be silently ignored if its destination address is a global IPv6 address.

In order to minimise the number of packets being sent while avoiding lower-layer fragmentation, a Babel node SHOULD attempt to maximise the size of the packets it sends, up to the outgoing interface's MTU adjusted for lower-layer headers (28 octets for UDP over IPv4, 48 octets for UDP over IPv6). It MUST NOT send packets larger than the attached interface's MTU adjusted for lower-layer headers or 512 octets, whichever is larger, but not exceeding 2^16 - 1 adjusted for lower-layer headers. Every Babel speaker MUST be able to receive packets that are as large as any attached interface's MTU adjusted for lower-layer headers or 512 octets, whichever is larger. Babel packets MUST NOT be sent in IPv6 Jumbograms.

In order to avoid global synchronisation of a Babel network and to aggregate multiple TLVs into large packets, a Babel node SHOULD buffer every TLV and delay sending a packet by a small, randomly chosen delay [JITTER]. In order to allow accurate computation of packet loss rates, this delay MUST NOT be larger than half the advertised Hello interval.

4.1. Data Types

4.1.1. Interval

Relative times are carried as 16-bit values specifying a number of centiseconds (hundredths of a second). This allows times up to roughly 11 minutes with a granularity of 10ms, which should cover all reasonable applications of Babel.

4.1.2. Router-Id

A router-id is an arbitrary 8-octet value. A router-id MUST NOT consist of either all zeroes or all ones.

4.1.3. Address

Since the bulk of the protocol is taken by addresses, multiple ways of encoding addresses are defined. Additionally, a common subnet prefix may be omitted when multiple addresses are sent in a single packet -- this is known as address compression (Section 4.6.9).

Address encodings:

- AE 0: wildcard address. The value is 0 octets long.

- AE 1: IPv4 address. Compression is allowed. 4 octets or less.

- AE 2: IPv6 address. Compression is allowed. 16 octets or less.

- AE 3: link-local IPv6 address. Compression is not allowed. The value is 8 octets long, a prefix of fe80::/64 is implied.

The address family associated to an address encoding is either IPv4 or IPv6; it is undefined for AE 0, IPv4 for AE 1, and IPv6 for AEs 2 and 3.

4.1.4. Prefixes

A network prefix is encoded just like a network address, but it is stored in the smallest number of octets that are enough to hold the significant bits (up to the prefix length).

4.2. Packet Format

A Babel packet consists of a 4-octet header, followed by a sequence of TLVs.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Magic     |    Version    |        Body length            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Packet Body ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-
```

Fields :

Magic   The arbitrary but carefully chosen value 42 (decimal); packets with a first octet different from 42 MUST be silently ignored.

Version   This document specifies version 2 of the Babel protocol. Packets with a second octet different from 2 MUST be silently ignored.

Body length The length in octets of the body following the packet header (excluding the Magic, Version and Body length fields).

Body   The packet body; a sequence of TLVs.

Any data following the body MUST be silently ignored.

4.3. TLV Format

With the exception of Pad1, all TLVs have the following structure:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Type      |    Length     |     Payload...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
```

Fields :

Type The type of the TLV.

Length The length of the body, exclusive of the Type and Length fields. If the body is longer than the expected length of a given type of TLV, any extra data MUST be silently ignored.

Payload The TLV payload, which consists of a body and, for selected TLV types, an optional list of sub-TLVs.

TLVs with an unknown type value MUST be silently ignored.

4.4. Sub-TLV Format

Every TLV carries an explicit length in its header; however, most TLVs are self-terminating, in the sense that it is possible to determine the length of the body without reference to the explicit Length field. If a TLV has a self-terminating format, then it MAY allow a sequence of sub-TLVs to follow the body.

Sub-TLVs have the same structure as TLVs. With the exception of PAD1, all TLVs have the following structure:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Type      |    Length     |     Body...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
```

Fields :

Type The type of the sub-TLV.

Length The length of the body, in octets, exclusive of the Type and Length fields.

Body The sub-TLV body, the interpretation of which depends on both the type of the sub-TLV and the type of the TLV within which it is embedded.

The most-significant bit of the sub-TLV, called the mandatory bit, indicates how to handle unknown sub-TLVs. If the mandatory bit is not set, then an unknown sub-TLV MUST be silently ignored, and the rest of the TLV processed normally. If the mandatory bit is set, then the whole enclosing TLV MUST be silently ignored (except for updating the parser state by a Router-Id, Next-Hop or Update TLV, see Section 4.6.7, Section 4.6.8, and Section 4.6.9).

4.5. Parser state

Babel uses a stateful parser: a TLV may refer to data from a previous TLV. The parser state consists of the following pieces of data:

- for each address encoding that allows compression, the current default prefix; this is undefined at the start of the packet, and is updated by each Update TLV with the Prefix flag set (Section 4.6.9);

- for each address family (IPv4 or IPv6), the current next-hop; this is the source address of the enclosing packet for the matching address family at the start of a packet, and is updated by each Next-Hop TLV (Section 4.6.8);

- the current router-id; this is undefined at the start of the packet, and is updated by each Router-ID TLV (Section 4.6.7) and by each Update TLV with Router-Id flag set.

Since the parser state is separate from the bulk of Babel's state, and since for correct parsing it must be identical across implementations, it is updated before checking for mandatory TLVs: parsing a TLV MUST update the parser state even if the TLV is otherwise ignored due to an unknown mandatory sub-TLV.

4.6. Details of Specific TLVs

4.6.1. Pad1

```
 0
 0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+
|   Type = 0    |
+-+-+-+-+-+-+-+-+
```

Fields :

Type   Set to 0 to indicate a Pad1 TLV.

This TLV is silently ignored on reception.

4.6.2. PadN

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Type = 1   |    Length     |      MBZ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
```

Fields :

Type   Set to 1 to indicate a PadN TLV.

Length   The length of the body, exclusive of the Type and Length fields.

MBZ   Set to 0 on transmission.

This TLV is silently ignored on reception.

4.6.3. Acknowledgment Request

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Type = 2   |    Length     |          Reserved             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Nonce             |          Interval             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

This TLV requests that the receiver send an Acknowledgment TLV within the number of centiseconds specified by the Interval field.

Fields :

Type   Set to 2 to indicate an Acknowledgment Request TLV.

Length   The length of the body, exclusive of the Type and Length fields.

Reserved Sent as 0 and MUST be ignored on reception.

Nonce   An arbitrary value that will be echoed in the receiver’s Acknowledgment TLV.

Interval A time interval in centiseconds after which the sender will assume that this packet has been lost. This MUST NOT be 0. The receiver MUST send an Acknowledgment TLV before this time has elapsed (with a margin allowing for propagation time).

This TLV is self-terminating, and allows sub-TLVs.

4.6.4. Acknowledgment

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Type = 3   |    Length     |            Nonce              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

This TLV is sent by a node upon receiving an Acknowledgment Request.

Fields :

Type   Set to 3 to indicate an Acknowledgment TLV.

Length   The length of the body, exclusive of the Type and Length fields.

Nonce   Set to the Nonce value of the Acknowledgment Request that prompted this Acknowledgment.

Since nonce values are not globally unique, this TLV MUST be sent to a unicast address.

This TLV is self-terminating, and allows sub-TLVs.

4.6.5. Hello

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Type = 4   |    Length     |            Flags              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Seqno             |           Interval            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

This TLV is used for neighbour discovery and for determining a neighbour's reception cost.

Fields :

Type   Set to 4 to indicate a Hello TLV.

Length   The length of the body, exclusive of the Type and Length fields.

Flags   The individual bits of this field specify special handling of this TLV (see below).

Seqno If the Unicast flag is set, this is the value of the sending node's outgoing Unicast Hello seqno for this neighbour. Otherwise, it is the sending node's outgoing Multicast Hello seqno for this interface.

Interval If non-zero, this is an upper bound, expressed in centiseconds, on the time after which the sending node will send a new scheduled Hello TLV with the same setting of the Unicast flag. If this is 0, then this Hello represents an unscheduled Hello, and doesn't carry any new information about times at which Hellos are sent.

The Flags field is interpreted as follows:

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|U|X|X|X|X|X|X|X|X|X|X|X|X|X|X|X|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- U (Unicast) flag (8000 hexadecimal): if set, then this Hello represents a Unicast Hello, otherwise it represents a Multicast Hello;

- X: all other bits MUST be sent as 0 and silently ignored on reception.

Every time a Hello is sent, the corresponding seqno counter MUST be incremented. Since there is a single seqno counter for all the Multicast Hellos sent by a given node over a given interface, if the Unicast flag is not set, this TLV MUST be sent to all neighbors on this link, which can be achieved by sending to a multicast destination, or by sending multiple packets to the unicast addresses of all reachable neighbours. Conversely, if the Unicast flag is set, this TLV MUST be sent to a single neighbour, which can achieved by sending to a unicast destination. In order to avoid large discontinuities in link quality, multiple Hello TLVs SHOULD NOT be sent in the same packet.

This TLV is self-terminating, and allows sub-TLVs.

4.6.6. IHU

```
    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |    Type = 5   |    Length     |       AE      |    Reserved   |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |            Rxcost             |           Interval            |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |       Address...
   +-+-+-+-+-+-+-+-+-+-+-+-+-
```

An IHU ("I Heard You") TLV is used for confirming bidirectional reachability and carrying a link’s transmission cost.

Fields :

Type   Set to 5 to indicate an IHU TLV.

Length   The length of the body, exclusive of the Type and Length fields.

AE   The encoding of the Address field. This should be 1 or 3 in most cases. As an optimisation, it MAY be 0 if the TLV is sent to a unicast address, if the association is over a point-to-point link, or when bidirectional reachability is ascertained by means outside of the Babel protocol.

Reserved Sent as 0 and MUST be ignored on reception.

Rxcost   The rxcost according to the sending node of the interface whose address is specified in the Address field. The value FFFF hexadecimal (infinity) indicates that this interface is unreachable.

Interval An upper bound, expressed in centiseconds, on the time after which the sending node will send a new IHU; this MUST NOT be 0. The receiving node will use this value in order to compute a hold time for this symmetric association.

Address   The address of the destination node, in the format specified by the AE field. Address compression is not allowed.

Conceptually, an IHU is destined to a single neighbour. However, IHU TLVs contain an explicit destination address, and MAY be sent to a multicast address, as this allows aggregation of IHUs destined to distinct neighbours into a single packet and avoids the need for an ARP or Neighbour Discovery exchange when a neighbour is not being used for data traffic.

IHU TLVs with an unknown value in the AE field MUST be silently ignored.

This TLV is self-terminating, and allows sub-TLVs.

4.6.7. Router-Id

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Type = 6   |    Length     |           Reserved            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                           Router-Id                           +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

A Router-Id TLV establishes a router-id that is implied by subsequent Update TLVs. This TLV sets the router-id even if it is otherwise ignored due to an unknown mandatory sub-TLV.

Fields :

Type   Set to 6 to indicate a Router-Id TLV.

Length   The length of the body, exclusive of the Type and Length fields.

Reserved Sent as 0 and MUST be ignored on reception.

Router-Id The router-id for routes advertised in subsequent Update TLVs. This MUST NOT consist of all zeroes or all ones.

This TLV is self-terminating, and allows sub-TLVs.

4.6.8. Next Hop

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Type = 7   |    Length     |      AE       |   Reserved    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       Next hop...
+-+-+-+-+-+-+-+-+-+-+-+-
```

A Next Hop TLV establishes a next-hop address for a given address family (IPv4 or IPv6) that is implied in subsequent Update TLVs.

This TLV sets up the next-hop for subsequent Update TLVs even if it is otherwise ignored due to an unknown mandatory sub-TLV.

Fields :

Type   Set to 7 to indicate a Next Hop TLV.

Length   The length of the body, exclusive of the Type and Length fields.

AE   The encoding of the Address field. This SHOULD be 1 or 3 and MUST NOT be 0.

Reserved Sent as 0 and MUST be ignored on reception.

Next hop The next-hop address advertised by subsequent Update TLVs, for this address family.

When the address family matches the network-layer protocol that this packet is transported over, a Next Hop TLV is not needed: in the absence of a Next Hop TLV in a given address family, the next hop address is taken to be the source address of the packet.

Next Hop TLVs with an unknown value for the AE field MUST be silently ignored.

This TLV is self-terminating, and allows sub-TLVs.

4.6.9. Update

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Type = 8   |    Length     |       AE      |    Flags      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Plen      |    Omitted    |            Interval           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Seqno             |            Metric             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      Prefix...
+-+-+-+-+-+-+-+-+-+-+-+-+-
```

An Update TLV advertises or retracts a route. As an optimisation, it can optionally have the side effect of establishing a new implied router-id and a new default prefix.

Fields :

Type Set to 8 to indicate an Update TLV.

Length The length of the body, exclusive of the Type and Length fields.

AE The encoding of the Prefix field.

Flags The individual bits of this field specify special handling of this TLV (see below).

Plen The length of the advertised prefix.

Omitted The number of octets that have been omitted at the beginning of the advertised prefix and that should be taken from a preceding Update TLV in the same address family with the Prefix flag set.

Interval An upper bound, expressed in centiseconds, on the time after which the sending node will send a new update for this prefix. This MUST NOT be 0. The receiving node will use this value to compute a hold time for the route table entry. The value FFFF hexadecimal (infinity) expresses that this announcement will not be repeated unless a request is received (Section 3.8.2.3).

Seqno The originator’s sequence number for this update.

Metric The sender’s metric for this route. The value FFFF hexadecimal (infinity) means that this is a route retraction.

Prefix The prefix being advertised. This field’s size is (Plen/8 - Omitted) rounded upwards.

The Flags field is interpreted as follows:

```
+-+-+-+-+-+-+-+-+
|P|R|X|X|X|X|X|X|
+-+-+-+-+-+-+-+-+
```

- P (Prefix) flag (80 hexadecimal): if set, then this Update establishes a new default prefix for subsequent Update TLVs with a matching address encoding within the same packet, even if this TLV is otherwise ignored due to an unknown mandatory sub-TLV;

- R (Router-Id) flag (40 hexadecimal): if set, then this TLV establishes a new default router-id for this TLV and subsequent Update TLVs in the same packet, even if this TLV is otherwise ignored due to an unknown mandatory sub-TLV. This router-id is computed from the first address of the advertised prefix as follows:

* if the length of the address is 8 octets or more, then the new router-id is taken from the 8 last octets of the address;

* if the length of the address is smaller than 8 octets, then the new router-id consists of the required number of zero octets followed by the address, i.e., the address is stored on the right of the router-id. For example, for an IPv4 address, the router-id consists of 4 octets of zeroes followed by the IPv4 address.

o X: all other bits MUST be sent as 0 and silently ignored on reception.

The prefix being advertised by an Update TLV is computed as follows:

o the first Omitted octets of the prefix are taken from the previous Update TLV with the Prefix flag set and the same address encoding, even if it was ignored due to an unknown mandatory sub-TLV;

o the next (Plen/8 - Omitted) rounded upwards octets are taken from the Prefix field;

o the remaining octets are set to 0. If AE is 3 (link-local IPv6), Omitted MUST be 0)

If the Metric field is finite, the router-id of the originating node for this announcement is taken from the prefix advertised by this Update if the Router-Id flag is set, computed as described above. Otherwise, it is taken either from the preceding Router-Id packet, or the preceding Update packet with the Router-Id flag set, whichever comes last, even if that TLV is otherwise ignored due to an unknown mandatory sub-TLV.

The next-hop address for this update is taken from the last preceding Next Hop TLV with a matching address family (IPv4 or IPv6) in the same packet even if it was otherwise ignored due to an unknown mandatory sub-TLV; if no such TLV exists, it is taken from the network-layer source address of this packet.

If the metric field is FFFF hexadecimal, this TLV specifies a retraction. In that case, the router-id, next-hop and seqno are not used. AE MAY then be 0, in which case this Update retracts all of the routes previously advertised by the sending interface. If the metric is finite, AE MUST NOT be 0. If the metric is infinite and AE is 0, Plen and Omitted MUST both be 0.

Update TLVs with an unknown value in the AE field MUST be silently ignored.

This TLV is self-terminating, and allows sub-TLVs.

4.6.10. Route Request

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Type = 9   |    Length     |      AE       |     Plen      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      Prefix...
+-+-+-+-+-+-+-+-+-+-+-+-+-
```

A Route Request TLV prompts the receiver to send an update for a given prefix, or a full route table dump.

Fields :

Type   Set to 9 to indicate a Route Request TLV.

Length   The length of the body, exclusive of the Type and Length fields.

AE   The encoding of the Prefix field. The value 0 specifies that this is a request for a full route table dump (a wildcard request).

Plen   The length of the requested prefix.

Prefix   The prefix being requested. This field’s size is Plen/8 rounded upwards.

A Request TLV prompts the receiver to send an update message (possibly a retraction) for the prefix specified by the AE, Plen, and Prefix fields, or a full dump of its route table if AE is 0 (in which case Plen MUST be 0 and Prefix is of length 0).

This TLV is self-terminating, and allows sub-TLVs.

4.6.11. Seqno Request

```
    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |    Type = 10  |     Length    |      AE       |     Plen      |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |             Seqno             |  Hop Count    |   Reserved    |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                                                               |
   +                          Router-Id                            +
   |                                                               |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |   Prefix...
   +-+-+-+-+-+-+-+-+-+-+-
```

A Seqno Request TLV prompts the receiver to send an Update for a given prefix with a given sequence number, or to forward the request further if it cannot be satisfied locally.

Fields :

Type   Set to 10 to indicate a Seqno Request message.

Length   The length of the body, exclusive of the Type and Length fields.

AE   The encoding of the Prefix field. This MUST NOT be 0.

Plen   The length of the requested prefix.

Seqno   The sequence number that is being requested.

Hop Count The maximum number of times that this TLV may be forwarded, plus 1. This MUST NOT be 0.

Reserved Sent as 0 and MUST be ignored on reception.

Router Id The Router-Id that is being requested. This MUST NOT consist of all zeroes or all ones.

Prefix   The prefix being requested. This field's size is Plen/8 rounded upwards.

A Seqno Request TLV prompts the receiving node to send a finite-metric Update for the prefix specified by the AE, Plen, and Prefix fields, with either a router-id different from what is specified by the Router-Id field, or a Seqno no less (modulo 2^16) than what is specified by the Seqno field. If this request cannot be satisfied locally, then it is forwarded according to the rules set out in Section 3.8.1.2.

While a Seqno Request MAY be sent to a multicast address, it MUST NOT be forwarded to a multicast address and MUST NOT be forwarded to more than one neighbour. A request MUST NOT be forwarded if its Hop Count field is 1.

This TLV is self-terminating, and allows sub-TLVs.

4.7. Details of specific sub-TLVs

4.7.1. Pad1

```
 0
 0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+
|   Type = 0    |
+-+-+-+-+-+-+-+-+
```

Fields :

Type Set to 0 to indicate a Pad1 sub-TLV.

This sub-TLV is silently ignored on reception. It is allowed within any TLV that allows sub-TLVs.

4.7.2. PadN

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Type = 1    |    Length     |      MBZ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
```

Fields :

Type Set to 1 to indicate a PadN sub-TLV.

Length The length of the body, in octets, exclusive of the Type and Length fields.

MBZ Set to 0 on transmission.

This sub-TLV is silently ignored on reception. It is allowed within any TLV that allows sub-TLVs.

5. IANA Considerations

IANA has registered the UDP port number 6696, called "babel", for use by the Babel protocol.

IANA has registered the IPv6 multicast group ff02::1:6 and the IPv4 multicast group 224.0.0.111 for use by the Babel protocol.

IANA has created a registry called "Babel TLV Types". The values in this registry are not changed by this specification.

IANA has created a registry called "Babel sub-TLV Types". Due to the addition of a Mandatory bit to the Babel protocol, the values in the "Babel sub-TLV Types" registry are amended as follows:

Type	Name	Reference
0	Pad1	this document
1	PadN	this document
112-126	Reserved for Experimental Use	this document
127	Reserved for expansion of the type space	this document
240-254	Reserved for Experimental Use	this document
255	Reserved for expansion of the type space	this document

Existing assignments in the "Babel sub-TLV Types" registry in the range 2 to 111 are not changed by this specification. The values 224 through 239, previously reserved for Experimental Use, are now unassigned.

IANA has created a registry called "Babel Flags Values". IANA is instructed to rename this registry to "Babel Update Flags Values", with its contents unchanged.

IANA is instructed to create a new registry called "Babel Hello Flags Values". The allocation policy for this registry is Specification Required [RFC5226]. The initial values in this registry are as follows:

Bit	Name	Reference
0	Unicast	this document
1-15	Unassigned	

IANA is instructed to replace all references to RFCs 6126 and 7557 in all of the registries mentioned above by references to this document.

6. Security Considerations

As defined in this document, Babel is a completely insecure protocol. Any attacker can misdirect data traffic by advertising routes with a low metric or a high seqno. This issue can be solved either by a lower-layer security mechanism (e.g. link-layer security), or by deploying a suitable authentication mechanism within Babel itself. With the exception of Hello TLVs used for discovery, Babel control traffic can be carried over unicast, which makes it possible to protect Babel traffic with a protocol that can only protect unicast data, for example IPsec with IKEv2, or DTLS.

The information that a Babel node announces to the whole routing domain is often sufficient to determine a mobile node's physical location with reasonable precision. The privacy issues that this causes can be mitigated somewhat by using randomly chosen router-ids and randomly chosen IP addresses, and changing them periodically.

When carried over IPv6, Babel packets are ignored unless they are sent from a link-local IPv6 address; since routers don't forward link-local IPv6 packets, this provides protection against spoofed Babel packets being sent from the global Internet. No such natural protection exists when Babel packets are carried over IPv4.

7. Acknowledgments

A number of people have contributed text and ideas to this specification. The authors are particularly indebted to Matthieu Boutier, Gwendoline Chouasne, Margaret Cullen, Donald Eastlake and Toke Hoiland-Jorgensen. The address compression technique was inspired by [PACKETBB].

8. References

8.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", RFC 2119, March 1997.

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.

8.2. Informative References

[DSDV] Perkins, C. and P. Bhagwat, "Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers", ACM SIGCOMM'94 Conference on Communications Architectures, Protocols and Applications 234-244, 1994.

[DUAL] Garcia Luna Aceves, J., "Loop-Free Routing Using Diffusing Computations", IEEE/ACM Transactions on Networking 1:1, February 1993.

[EIGRP] Albrightson, B., Garcia Luna Aceves, J., and J. Boyle, "EIGRP -- a Fast Routing Protocol Based on Distance Vectors", Proc. Interop 94, 1994.

[ETX] De Couto, D., Aguayo, D., Bicket, J., and R. Morris, "A high-throughput path metric for multi-hop wireless networks", Proc. MobiCom 2003, 2003.

[IS-IS] "Information technology -- Telecommunications and information exchange between systems -- Intermediate System to Intermediate System intra-domain routeing information exchange protocol for use in conjunction with the protocol for providing the connectionless-mode network service (ISO 8473)", ISO/IEC 10589:2002, 2002.

[JITTER] Floyd, S. and V. Jacobson, "The synchronization of periodic routing messages", IEEE/ACM Transactions on Networking 2, 2, 122-136, April 1994.

[OSPF] Moy, J., "OSPF Version 2", RFC 2328, April 1998.

[PACKETBB]
Clausen, T., Dearlove, C., Dean, J., and C. Adjih, "Generalized Mobile Ad Hoc Network (MANET) Packet/Message Format", RFC 5444, February 2009.

[RIP] Malkin, G., "RIP Version 2", RFC 2453, November 1998.

Appendix A. Cost and Metric Computation

The strategy for computing link costs and route metrics is a local matter; Babel itself only requires that it comply with the conditions given in Section 3.4.3 and Section 3.5.2. Different nodes may use different strategies in a single network and may use different strategies on different interface types. This section describes the strategies used by the sample implementation of Babel.

The sample implementation of Babel sends periodic Multicast Hellos, and never sends Unicast Hellos. It maintains statistics about the last 16 received Hello TLVs of each kind (Appendix A.1), computes costs by using the 2-out-of-3 strategy (Appendix A.2.1) on wired links, and ETX (Appendix A.2.2) on wireless links. It uses an additive algebra for metric computation (Appendix A.3.1).

A.1. Maintaining Hello History

For each neighbour, the sample implementation of Babel maintains two sets of Hello history, one for each kind of Hello, and an expected sequence number, one for Multicast and one for Unicast Hellos. Each Hello history is a vector of 16 bits, where a 1 value represents a received Hello, and a 0 value a missed Hello. For each kind of Hello, the expected sequence number, written ne, is the sequence number that is expected to be carried by the next received Hello from this neighbour.

Whenever it receives a Hello packet of a given kind from a neighbour, a node compares the received sequence number nr for that kind of Hello with its expected sequence number ne. Depending on the outcome of this comparison, one of the following actions is taken:

- if the two differ by more than 16 (modulo 2^16), then the sending node has probably rebooted and lost its sequence number; the whole associated neighbour table entry is flushed and a new one is created;

- otherwise, if the received nr is smaller (modulo 2^16) than the expected sequence number ne, then the sending node has increased its Hello interval without us noticing; the receiving node removes the last (ne - nr) entries from this neighbour's Hello history (we "undo history");

- otherwise, if nr is larger (modulo 2^16) than ne, then the sending node has decreased its Hello interval, and some Hellos were lost; the receiving node adds (nr - ne) 0 bits to the Hello history (we "fast-forward").

The receiving node then appends a 1 bit to the Hello history and sets ne to (nr + 1). If the Interval field of the received Hello is not zero, it resets the neighbour's hello timer to 1.5 times the advertised Interval (the extra margin allows for delay due to jitter).

Whenever either Hello timer associated to a neighbour expires, the local node adds a 0 bit to this neighbour's Hello history, and increments the expected Hello number. If both Hello histories are empty (they contain 0 bits only), the neighbour entry is flushed; otherwise, the relevant hello timer is reset to the value advertised in the last Hello of that kind received from this neighbour (no extra margin is necessary in this case, since jitter was already taken into account when computing the timeout that has just expired).

A.2. Cost Computation

This section discusses how to compute costs based on Hello history.

A.2.1. k-out-of-j

K-out-of-j link sensing is suitable for wired links that are either up, in which case they only occasionally drop a packet, or down, in which case they drop all packets.

The k-out-of-j strategy is parameterised by two small integers k and j, such that 0 < k <= j, and the nominal link cost, a constant K >= 1. A node keeps a history of the last j hellos; if k or more of those have been correctly received, the link is assumed to be up, and the rxcost is set to K; otherwise, the link is assumed to be down, and the rxcost is set to infinity.

Since Babel supports two kinds of Hellos, a Babel node performs k-out-of-j twice for each neighbour, once on the Unicast and once on the Multicast Hello history. If either of the instances of k-out-of-j indicates that the link is up, then the link is assumed to be up, and the rxcost is set to K; if both instances indicate that the link is down, then the link is assumed to be down, and the rxcost is set to infinity. In other words, the resulting rxcost is the minimum of the rxcosts yielded by the two instances of k-out-of-j link sensing.

The cost of a link performing k-out-of-j link sensing is defined as follows:

- cost = FFFF hexadecimal if rxcost = FFFF hexadecimal;

- cost = txcost otherwise.

A.2.2. ETX

Unlike wired links, which are bimodal (either up or down), wireless links exhibit continuous variation of the link quality. Naive application of hop-count routing in networks that use wireless links for transit tends to select long, lossy links in preference to shorter, lossless links, which can dramatically reduce throughput. For that reason, a routing protocol designed to support wireless links must perform some form of link-quality estimation.

ETX [ETX] is a simple link-quality estimation algorithm that is designed to work well with the IEEE 802.11 MAC. By default, the IEEE 802.11 MAC performs ARQ and rate adaptation on unicast frames, but not on multicast frames, which are sent at a fixed rate with no ARQ; therefore, measuring the loss rate of multicast frames yields a useful estimate of a link's quality.

A node performing ETX link quality estimation uses a neighbour's Multicast Hello history to compute an estimate, written beta, of the probability that a Hello TLV is successfully received. Beta can be computed as the fraction of 1 bits within a small number (say, 6) of the most recent entries in the Multicast Hello history, or it can be an exponential average, or some combination of both approaches.

Let alpha be MIN(1, 256/txcost), an estimate of the probability of successfully sending a Hello TLV. The cost is then computed by

    cost = 256/(alpha * beta)

or, equivalently,

    cost = (MAX(txcost, 256) * rxcost) / 256.

Since the IEEE 802.11 MAC performs ARQ on unicast frames, unicast frames do not provide a useful measure of link quality, and therefore ETX ignores the Unicast Hello history. Thus, a node performing ETX link-quality estimation will not route through neighbouring nodes unless they send periodic Multicast Hellos (possibly in addition to Unicast Hellos).

A.3. Metric Computation

As described in Section 3.5.2, the metric advertised by a neighbour is combined with the link cost to yield a metric.

A.3.1. Additive Metrics

The simplest approach for obtaining a monotonic, isotonic metric is to define the metric of a route as the sum of the costs of the component links. More formally, if a neighbour advertises a route with metric m over a link with cost c, then the resulting route has metric M(c, m) = c + m.

A multiplicative metric can be converted into an additive one by taking the logarithm (in some suitable base) of the link costs.

A.3.2. External Sources of Willingness

A node may want to vary its willingness to forward packets by taking into account information that is external to the Babel protocol, such as the monetary cost of a link, the node's battery status, CPU load, etc. This can be done by adding to every route's metric a value k that depends on the external data. For example, if a battery-powered node receives an update with metric m over a link with cost c, it might compute a metric M(c, m) = k + c + m, where k depends on the battery status.

In order to preserve strict monotonicity (Section 3.5.2), the value k must be greater than -c.

Appendix B. Constants

The choice of time constants is a trade-off between fast detection of mobility events and protocol overhead. Two implementations of Babel with different time constants will interoperate, although the resulting convergence time will most likely be dictated by the slower of the two.

Experience with the sample implementation of Babel indicates that the Hello interval is the most important time constant: a mobility event is detected within 1.5 to 3 Hello intervals. Due to Babel's reliance on triggered updates and explicit requests, the Update interval only has an effect on the time it takes for accurate metrics to be propagated after variations in link costs too small to trigger an unscheduled update or in the presence of packet loss.

At the time of writing, the sample implementation of Babel uses the following default values:

Multicast Hello Interval: 4 seconds.

IHU Interval: the advertised IHU interval is always 3 times the Multicast Hello interval. IHUs are actually sent with each Hello on lossy links (as determined from the Hello history), but only with every third Multicast Hello on lossless links.

Unicast Hello Interval: the sample implementation never sends scheduled Unicast Hellos;

Update Interval: 4 times the Multicast Hello interval.

IHU Hold Time: 3.5 times the advertised IHU interval.

Route Expiry Time: 3.5 times the advertised update interval.

Source GC time: 3 minutes.

Request timeout: initially 2 seconds, doubled every time a request is resent, up to a maximum of three times.

The amount of jitter applied to a packet depends on whether it contains any urgent TLVs or not (Section 3.1). Urgent triggered updates and urgent requests are delayed by no more than 200ms; acknowledgments, by no more than the associated deadline; and other TLVs by no more than one-half the Multicast Hello interval.

Appendix C. Considerations for protocol extensions

Babel is an extensible protocol, and this document defines a number of mechanisms that can be used to extend the protocol in a backwards compatible manner:

- increasing the version number in the packet header;
- defining new TLVs;
- defining new sub-TLVs (with or without the mandatory bit set);
- defining new AEs;
- using the packet trailer.

This appendix is intended to guide designers of protocol extensions in chosing a particular encoding.

The version number in the Babel header should only be increased if the new version is not backwards compatible with the original protocol.

In many cases, an extension could be implemented either by defining a new TLV, or by adding a new sub-TLV to an existing TLV. For example, an extension whose purpose is to attach additional data to route updates can be implemented either by creating a new "enriched" Update TLV, by adding a non-mandatory sub-TLV to the Update TLV, or by adding a mandatory sub-TLV.

The various encodings are treated differently by implementations that do not understand the extension. In the case of a new TLV or of a sub-TLV with the mandatory bit set, the whole TLV is ignored by implementations that do not implement the extension, while in the case of a non-mandatory sub-TLV, the TLV is parsed and acted upon, and only the unknown sub-TLV is silently ignored. Therefore, a non-mandatory sub-TLV should be used by extensions that extend the Update in a compatible manner (the extension data may be silently ignored), while a mandatory sub-TLV or a new TLV must be used by extensions that make incompatible extensions to the meaning of the TLV (the whole TLV must be thrown away if the extension data is not understood).

Experience shows that the need for additional data tends to crop up in the most unexpected places. Hence, it is recommended that extensions that define new TLVs should make them self-terminating, and allow attaching sub-TLVs to them.

Adding a new AE is essentially equivalent to adding a new TLV: Update TLVs with an unknown AE are ignored, just like unknown TLVs. However, adding a new AE is more involved than adding a new TLV, since it creates a new set of compression state. Additionally, since the Next Hop TLV creates state specific to a given address family, as opposed to a given AE, a new AE for a previously defined address family must not be used in the Next Hop TLV if backwards compatibility is required. A similar issue arises with Update TLVs with unknown AEs establishing a new router-id (due to the Router-Id flag being set). Therefore, defining new AEs must be done with care if compatibility with unextended implementations is required.

The packet trailer (the space after the declared length of the packet but within the payload of the UDP datagram) was originally intended to carry a cryptographic signature. However, no extension has used it to date, and therefore we refrain from making any recommendations about its use due to the lack of implementation experience.

Appendix D. Stub Implementations

Babel is a fairly economic protocol. Updates take between 12 and 40 octets per destination, depending on the address family and how successful compression is; in a double-stack flat network, an average of less than 24 octets per update is typical. The route table occupies about 35 octets per IPv6 entry. To put these values into perspective, a single full-size Ethernet frame can carry some 65 route updates, and a megabyte of memory can contain a 20000-entry route table and the associated source table.

Babel is also a reasonably simple protocol. The sample implementation consists of less than 12 000 lines of C code, and it compiles to less than 120 kB of text on a 32-bit CISC architecture; about half of this figure is due to protocol extensions and user-interface code.

Nonetheless, in some very constrained environments, such as PDAs, microwave ovens, or abacuses, it may be desirable to have subset implementations of the protocol.

There are many different definitions of a stub router, but for the needs of this section a stub implementation of Babel is one that announces one or more directly attached prefixes into a Babel network but doesn’t reannounce any routes that it has learnt from its neighbours. It may either maintain a full routing table, or simply select a default gateway amongst any one of its neighbours that announces a default route. Since a stub implementation never forwards packets except from or to directly attached links, it cannot possibly participate in a routing loop, and hence it need not evaluate the feasibility condition or maintain a source table.

No matter how primitive, a stub implementation MUST parse sub-TLVs attached to any TLVs that it understands and check the mandatory bit. It MUST answer acknowledgment requests and MUST participate in the Hello/IHU protocol. It MUST also be able to reply to seqno requests for routes that it announces and SHOULD be able to reply to route requests.

Experience shows that an IPv6-only stub implementation of Babel can be written in less than 1000 lines of C code and compile to 13 kB of text on 32-bit CISC architecture.

Appendix E. Software Availability

The sample implementation of Babel is available from <https://www.irif.fr/˜jch/software/babel/>.

Appendix F. Changes from previous versions

F.1. Changes since RFC 6126

o Changed UDP port number to 6696.

o Consistently use router-id rather than id.

- Clarified that the source garbage collection timer is reset after sending an update even if the entry was not modified.

- In section "Seqno Requests", fixed an erroneous "route request".

- In the description of the Seqno Request TLV, added the description of the Router-Id field.

- Made router-ids all-0 and all-1 forbidden.

F.2. Changes since draft-ietf-babel-rfc6126bis-00

- Added security considerations.

F.3. Changes since draft-ietf-babel-rfc6126bis-01

- Integrated the format of sub-TLVs.

- Mentioned for each TLV whether it supports sub-TLVs.

- Added Appendix C.

- Added a mandatory bit in sub-TLVs.

- Changed compression state to be per-AF rather than per-AE.

- Added implementation hint for the routing table.

- Clarified how router-ids are computed when bit 0x40 is set in Updates.

- Relaxed the conditions for sending requests, and tightened the conditions for forwarding requests.

- Clarified that neighbours should be acquired at some point, but it doesn't matter when.

F.4. Changes since draft-ietf-babel-rfc6126bis-02

- Added Unicast Hellos.

- Added unscheduled (interval-less) Hellos.

- Changed Appendix A to consider Unicast and unscheduled Hellos.

- Changed Appendix B to agree with the reference implementation.

- Added optional algorithm to avoid the hold time.

- Changed the table of pending seqno requests to be indexed by router-id in addition to prefixes.

- Relaxed the route acquisition algorithm.

- Replaced minimal implementations by stub implementations.

- Added acknowledgments section.

F.5. Changes since draft-ietf-babel-rfc6126bis-03

- Clarified that all the data structures are conceptual.

- Made sending and receiving Multicast Hellos a SHOULD, avoids expressing any opinion about Unicast Hellos.

- Removed opinion about Multicast vs. Unicast Hellos (Appendix A.4).

- Made hold-time into a SHOULD rather than MUST.

- Clarified that Seqno Requests are for a finite-metric Update.

- Clarified that sub-TLVs Pad1 and PadN are allowed within any TLV that allows sub-TLVs.

- Updated IANA Considerations.

- Updated Security Considerations.

- Renamed routing table back to route table.

- Made buffering outgoing updates a SHOULD.

- Weakened advice to use modified EUI-64 in router-ids.

- Added information about sending requests to Appendix B.

- A number of minor wording changes and clarifications.

Authors' Addresses

Juliusz Chroboczek
IRIF, University of Paris-Diderot
Case 7014
75205 Paris Cedex 13
France

Email: jch@irif.fr

David Schinazi
Apple Inc.
1 Infinite Loop
Cupertino, California 95014
US

Email: dschinazi@apple.com

Network Working Group
Internet-Draft
Obsoletes: 6126,7557 (if approved)
Intended status: Standards Track
Expires: May 18, 2019

J. Chroboczek
IRIF, University of Paris-Diderot
D. Schinazi
Google LLC

November 14, 2018

The Babel Routing Protocol
draft-ietf-babel-rfc6126bis-07

Abstract

Babel is a loop-avoiding distance-vector routing protocol that is robust and efficient both in ordinary wired networks and in wireless mesh networks. This document describes the Babel routing protocol, and obsoletes RFCs 6126 and 7557

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

Babel is a loop-avoiding distance-vector routing protocol that is designed to be robust and efficient both in networks using prefix-based routing and in networks using flat routing ("mesh networks"), and both in relatively stable wired networks and in highly dynamic wireless networks.

1.1. Features

The main property that makes Babel suitable for unstable networks is that, unlike naive distance-vector routing protocols [RIP], it strongly limits the frequency and duration of routing pathologies such as routing loops and black-holes during reconvergence. Even after a mobility event is detected, a Babel network usually remains loop-free. Babel then quickly reconverges to a configuration that preserves the loop-freedom and connectedness of the network, but is not necessarily optimal; in many cases, this operation requires no packet exchanges at all. Babel then slowly converges, in a time on the scale of minutes, to an optimal configuration. This is achieved by using sequenced routes, a technique pioneered by Destination-Sequenced Distance-Vector routing [DSDV].

More precisely, Babel has the following properties:

- when every prefix is originated by at most one router, Babel never suffers from routing loops;

- when a single prefix is originated by multiple routers, Babel may occasionally create a transient routing loop for this particular prefix; this loop disappears in a time proportional to its diameter, and never again (up to an arbitrary garbage-collection (GC) time) will the routers involved participate in a routing loop for the same prefix;

- assuming bounded packet loss rates, any routing black-holes that may appear after a mobility event are corrected in a time at most proportional to the network's diameter.

Babel has provisions for link quality estimation and for fairly arbitrary metrics. When configured suitably, Babel can implement shortest-path routing, or it may use a metric based, for example, on measured packet loss.

Babel nodes will successfully establish an association even when they are configured with different parameters. For example, a mobile node that is low on battery may choose to use larger time constants (hello and update intervals, etc.) than a node that has access to wall power. Conversely, a node that detects high levels of mobility may choose to use smaller time constants. The ability to build such heterogeneous networks makes Babel particularly adapted to the unmanaged and wireless environment.

Finally, Babel is a hybrid routing protocol, in the sense that it can carry routes for multiple network-layer protocols (IPv4 and IPv6), whichever protocol the Babel packets are themselves being carried over.

1.2. Limitations

Babel has two limitations that make it unsuitable for use in some environments. First, Babel relies on periodic routing table updates rather than using a reliable transport; hence, in large, stable networks it generates more traffic than protocols that only send updates when the network topology changes. In such networks, protocols such as OSPF [OSPF], IS-IS [IS-IS], or the Enhanced Interior Gateway Routing Protocol (EIGRP) [EIGRP] might be more suitable.

Second, unless the optional algorithm described in Section 3.5.5 is implemented, Babel does impose a hold time when a prefix is retracted. While this hold time does not apply to the exact prefix being retracted, and hence does not prevent fast reconvergence should it become available again, it does apply to any shorter prefix that covers it. This may make those implementations of Babel that do not implement the optional algorithm described in Section 3.5.5 unsuitable for use in networks that implement automatic prefix aggregation.

1.3. Specification of Requirements

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2. Conceptual Description of the Protocol

Babel is a loop-avoiding distance vector protocol: it is based on the Bellman-Ford protocol, just like the venerable RIP [RIP], but includes a number of refinements that either prevent loop formation altogether, or ensure that a loop disappears in a timely manner and doesn’t form again.

Conceptually, Bellman-Ford is executed in parallel for every source of routing information (destination of data traffic). In the following discussion, we fix a source S; the reader will recall that the same algorithm is executed for all sources.

2.1. Costs, Metrics and Neighbourship

For every pair of neighbouring nodes A and B, Babel computes an abstract value known as the cost of the link from A to B., written C(A, B). Given a route between any two (not necessarily neighbouring) nodes, the metric of the route is the sum of the costs of all the edges along the route. The goal of the routing algorithm is to compute, for every source S, the tree of routes of lowest metric to S.

Costs and metrics need not be integers. In general, they can be values in any algebra that satisfies two fairly general conditions (Section 3.5.2).

A Babel node periodically sends Hello messages to all of its neighbours; it also periodically sends an IHU ("I Heard You") message to every neighbour from which it has recently heard a Hello. From the information derived from Hello and IHU messages received from its neighbour B, a node A computes the cost C(A, B) of the link from A to B.

2.2. The Bellman-Ford Algorithm

Every node A maintains two pieces of data: its estimated distance to S, written D(A), and its next-hop router to S, written NH(A). Initially, D(S) = 0, D(A) is infinite, and NH(A) is undefined.

Periodically, every node B sends to all of its neighbours a route update, a message containing D(B). When a neighbour A of B receives the route update, it checks whether B is its selected next hop; if that is the case, then NH(A) is set to B, and D(A) is set to C(A, B) + D(B). If that is not the case, then A compares C(A, B) + D(B) to its current value of D(A). If that value is smaller, meaning that the received update advertises a route that is better than the currently selected route, then NH(A) is set to B, and D(A) is set to C(A, B) + D(B).

A number of refinements to this algorithm are possible, and are used by Babel. In particular, convergence speed may be increased by sending unscheduled "triggered updates" whenever a major change in the topology is detected, in addition to the regular, scheduled updates. Additionally, a node may maintain a number of alternate routes, which are being advertised by neighbours other than its selected neighbour, and which can be used immediately if the selected route were to fail.

2.3. Transient Loops in Bellman-Ford

It is well known that a naive application of Bellman-Ford to distributed routing can cause transient loops after a topology change. Consider for example the following topology:

```
         B
      1 /|
   1   / |
S --- A  |1
       \ |
      1 \|
         C
```

After convergence, D(B) = D(C) = 2, with NH(B) = NH(C) = A.

Suppose now that the link between S and A fails:

```
         B
      1 /|
       / |
S     A  |1
       \ |
      1 \|
         C
```

When it detects the failure of the link, A switches its next hop to B (which is still advertising a route to S with metric 2), and advertises a metric equal to 3, and then advertises a new route with metric 3. This process of nodes changing selected neighbours and increasing their metric continues until the advertised metric reaches "infinity", a value larger than all the metrics that the routing protocol is able to carry.

2.4. Feasibility Conditions

Bellman-Ford is a very robust algorithm: its convergence properties are preserved when routers delay route acquisition or when they discard some updates. Babel routers discard received route announcements unless they can prove that accepting them cannot possibly cause a routing loop.

More formally, we define a condition over route announcements, known as the "feasibility condition", that guarantees the absence of routing loops whenever all routers ignore route updates that do not satisfy the feasibility condition. In effect, this makes Bellman-Ford into a family of routing algorithms, parameterised by the feasibility condition.

Many different feasibility conditions are possible. For example, BGP can be modelled as being a distance-vector protocol with a (rather drastic) feasibility condition: a routing update is only accepted when the receiving node's AS number is not included in the update's AS-Path attribute (note that BGP's feasibility condition does not ensure the absence of transient "micro-loops" during reconvergence).

Another simple feasibility condition, used in the Destination-Sequenced Distance-Vector (DSDV) routing protocol [DSDV] and in the Ad hoc On-Demand Distance Vector (AODV) protocol, stems from the following observation: a routing loop can only arise after a router has switched to a route with a larger metric than the route that it had previously selected. Hence, one could decide that a route is feasible only when its metric at the local node would be no larger than the metric of the currently selected route, i.e., an announcement carrying a metric D(B) is accepted by A when C(A, B) + D(B) <= D(A). If all routers obey this constraint, then the metric at every router is nonincreasing, and the following invariant is always preserved: if A has selected B as its successor, then D(B) < D(A), which implies that the forwarding graph is loop-free.

Babel uses a slightly more refined feasibility condition, derived from EIGRP [DUAL]. Given a router A, define the feasibility distance of A, written FD(A), as the smallest metric that A has ever advertised for S to any of its neighbours. An update sent by a neighbour B of A is feasible when the metric D(B) advertised by B is strictly smaller than A's feasibility distance, i.e., when D(B) < FD(A).

It is easy to see that this latter condition is no more restrictive than DSDV-feasibility. Suppose that node A obeys DSDV-feasibility; then D(A) is nonincreasing, hence at all times D(A) <= FD(A). Suppose now that A receives a DSDV-feasible update that advertises a metric D(B). Since the update is DSDV-feasible, C(A, B) + D(B) <= D(A), hence D(B) < D(A), and since D(A) <= FD(A), D(B) < FD(A).

To see that it is strictly less restrictive, consider the following diagram, where A has selected the route through B, and D(A) = FD(A) = 2. Since D(C) = 1 < FD(A), the alternate route through C is feasible for A, although its metric C(A, C) + D(C) = 5 is larger than that of the currently selected route:

```
      B
   1 / \ 1
    /   \
   S     A
    \   /
   1 \ / 4
      C
```

To show that this feasibility condition still guarantees loop-freedom, recall that at the time when A accepts an update from B, the metric D(B) announced by B is no smaller than FD(B); since it is smaller than FD(A), at that point in time FD(B) < FD(A). Since this property is preserved when A sends updates, it remains true at all times, which ensures that the forwarding graph has no loops.

2.5. Solving Starvation: Sequencing Routes

Obviously, the feasibility conditions defined above cause starvation when a router runs out of feasible routes. Consider the following diagram, where both A and B have selected the direct route to S:

```
      A
   1 /|        D(A) = 1
    / |       FD(A) = 1
   S  |1
    \ |        D(B) = 2
   2 \|       FD(B) = 2
      B
```

Suppose now that the link between A and S breaks:

```
      A
      |
      |       FD(A) = 1
   S  |1
    \ |        D(B) = 2
   2 \|       FD(B) = 2
      B
```

The only route available from A to S, the one that goes through B, is not feasible: A suffers from spurious starvation. At that point, the whole subtree suffering from starvation must be reset, which is essentially what EIGRP does when it performs a global synchronisation of all the routers in the sarving subtree (the "active" phase of EIGRP).

Babel reacts to starvation in a less drastic manner, by using sequenced routes, a technique introduced by DSDV and adopted by AODV. In addition to a metric, every route carries a sequence number, a nondecreasing integer that is propagated unchanged through the network and is only ever incremented by the source; a pair (s, m), where s is a sequence number and m a metric, is called a distance.

A received update is feasible when either it is more recent than the feasibility distance maintained by the receiving node, or it is equally recent and the metric is strictly smaller. More formally, if FD(A) = (s, m), then an update carrying the distance (s’, m’) is feasible when either s’ > s, or s = s’ and m’ < m.

Assuming the sequence number of S is 137, the diagram above becomes:

```
   A
   |
   |       FD(A) = (137, 1)
S  |1
 \ |        D(B) = (137, 2)
2 \|       FD(B) = (137, 2)
   B
```

After S increases its sequence number, and the new sequence number is propagated to B, we have:

```
   A
   |
   |       FD(A) = (137, 1)
S  |1
 \ |        D(B) = (138, 2)
2 \|       FD(B) = (138, 2)
   B
```

at which point the route through B becomes feasible again.

Note that while sequence numbers are used for determining feasibility, they are not necessarily used in route selection: a node will normally ignore the sequence number when selecting the best route to a given destination (Section 3.6).

2.6. Requests

In DSDV, the sequence number of a source is increased periodically. A route becomes feasible again after the source increases its sequence number, and the new sequence number is propagated through the network, which may, in general, require a significant amount of time.

Babel takes a different approach. When a node detects that it is suffering from a potentially spurious starvation, it sends an explicit request to the source for a new sequence number. This request is forwarded hop by hop to the source, with no regard to the feasibility condition. Upon receiving the request, the source increases its sequence number and broadcasts an update, which is forwarded to the requesting node.

Note that after a change in network topology not all such requests will, in general, reach the source, as some will be sent over links that are now broken. However, if the network is still connected, then at least one among the nodes suffering from spurious starvation has an (unfeasible) route to the source; hence, in the absence of packet loss, at least one such request will reach the source. (Resending requests a small number of times compensates for packet loss.)

Since requests are forwarded with no regard to the feasibility condition, they may, in general, be caught in a forwarding loop; this is avoided by having nodes perform duplicate detection for the requests that they forward.

2.7. Multiple Routers

The above discussion assumes that every prefix is originated by a single router. In real networks, however, it is often necessary to have a single prefix originated by multiple routers: for example, the default route will be originated by all of the edge routers of a routing domain.

Since synchronising sequence numbers between distinct routers is problematic, Babel treats routes for the same prefix as distinct entities when they are originated by different routers: every route announcement carries the router-id of its originating router, and feasibility distances are not maintained per prefix, but per source, where a source is a pair of a router-id and a prefix. In effect, Babel guarantees loop-freedom for the forwarding graph to every source; since the union of multiple acyclic graphs is not in general acyclic, Babel does not in general guarantee loop-freedom when a prefix is originated by multiple routers, but any loops will be broken in a time at most proportional to the diameter of the loop -- as soon as an update has "gone around" the routing loop.

Consider for example the following topology, where A has selected the default route through S, and B has selected the one through S':

```
           1     1     1
::/0 -- S --- A --- B --- S' -- ::/0
```

Suppose that both default routes fail at the same time; then nothing prevents A from switching to B, and B simultaneously switching to A. However, as soon as A has successfully advertised the new route to B, the route through A will become unfeasible for B. Conversely, as soon as B will have advertised the route through A, the route through B will become unfeasible for A.

In effect, the routing loop disappears at the latest when routing information has gone around the loop. Since this process can be delayed by lost packets, Babel makes certain efforts to ensure that updates are sent reliably after a router-id change (Section 3.7.2).

Additionally, after the routers have advertised the two routes, both sources will be in their source tables, which will prevent them from ever again participating in a routing loop involving routes from S and S' (up to the source GC time, which, available memory permitting, can be set to arbitrarily large values).

2.8. Overlapping Prefixes

In the above discussion, we have assumed that all prefixes are disjoint, as is the case in flat ("mesh") routing. In practice, however, prefixes may overlap: for example, the default route overlaps with all of the routes present in the network.

After a route fails, it is not correct in general to switch to a route that subsumes the failed route. Consider for example the following configuration:

```
           1     1
::/0 -- A --- B --- C
```

Suppose that node C fails. If B forwards packets destined to C by following the default route, a routing loop will form, and persist until A learns of B's retraction of the direct route to C. B avoids this pitfall by installing an "unreachable" route after a route is retracted; this route is maintained until it can be guaranteed that the former route has been retracted by all of B's neighbours (Section 3.5.5).

3. Protocol Operation

Every Babel speaker is assigned a router-id, which is an arbitrary string of 8 octets that is assumed unique across the routing domain. For example, routers-ids could be assigned randomly, or they could derived from a link-layer address. (The protocol encoding is slightly more compact when router-ids are assigned in the same manner as the IPv6 layer assigns host IDs.)

3.1. Message Transmission and Reception

Babel protocol packets are sent in the body of a UDP datagram (as described in Section 4 below). Each Babel packet consists of zero or more TLVs. Most TLVs may contain sub-TLVs.

The source address of a Babel packet is always a unicast address, link-local in the case of IPv6. Babel packets may be sent to a well-known (link-local) multicast address or to a (link-local) unicast address. In normal operation, a Babel speaker sends both multicast and unicast packets to its neighbours.

With the exception of Hello TLVs and acknowledgments, all Babel TLVs can be sent to either unicast or multicast addresses, and their semantics does not depend on whether the destination is a unicast or a multicast address. Hence, a Babel speaker does not need to determine the destination address of a packet that it receives in order to interpret it.

A moderate amount of jitter may be applied to packets sent by a Babel speaker: outgoing TLVs are buffered and SHOULD be sent with a small random delay. This is done for two purposes: it avoids synchronisation of multiple Babel speakers across a network [JITTER], and it allows for the aggregation of multiple TLVs into a single packet.

The exact delay and amount of jitter applied to a packet depends on whether it contains any urgent TLVs. Acknowledgment TLVs MUST be sent before the deadline specified in the corresponding request. The particular class of updates specified in Section 3.7.2 MUST be sent in a timely manner. The particular class of request and update TLVs specified in Section 3.8.2 SHOULD be sent in a timely manner.

3.2. Data Structures

In this section, we give a description of the data structures that every Babel speaker maintains. This description is conceptual: a Babel speaker may use different data structures as long as the resulting protocol is the same as the one described in this document.

For example, rather than maintaining a single table containing both selected and unselected (fallback) routes, as described in Section 3.2.6 belong, an actual implementation would probably use two tables, one with selected routes and one with fallback routes.

3.2.1. Sequence number arithmetic

Sequence numbers (seqnos) appear in a number of Babel data structures, and they are interpreted as integers modulo 2^16. For the purposes of this document, arithmetic on sequence numbers is defined as follows.

Given a seqno s and an integer n, the sum of s and n is defined by

```
s + n (modulo 2^16) = (s + n) MOD 2^16
```

or, equivalently,

```
s + n (modulo 2^16) = (s + n) AND 65535
```

where MOD is the modulo operation yielding a non-negative integer and AND is the bitwise conjunction operation.

Given two sequence numbers s and s', the relation s is less than s' (s < s') is defined by

```
s < s' (modulo 2^16) when 0 < ((s' - s) MOD 2^16) < 32768
```

or equivalently

```
s < s' (modulo 2^16) when s /= s' and ((s' - s) AND 32768) = 0.
```

3.2.2. Node Sequence Number

A node's sequence number is a 16-bit integer that is included in route updates sent for routes originated by this node.

A node increments its sequence number (modulo 2^16) whenever it receives a request for a new sequence number (Section 3.8.1.2). A node SHOULD NOT increment its sequence number (seqno) spontaneously, since increasing seqnos makes it less likely that other nodes will have feasible alternate routes when their selected routes fail.

3.2.3. The Interface Table

The interface table contains the list of interfaces on which the node speaks the Babel protocol. Every interface table entry contains the interface's outgoing Multicast Hello seqno, a 16-bit integer that is sent with each Multicast Hello TLV on this interface and is incremented (modulo 2^16) whenever a Multicast Hello is sent. (Note that an interface's Multicast Hello seqno is unrelated to the node's seqno.)

There are two timers associated with each interface table entry -- the multicast hello timer, which governs the sending of scheduled Multicast Hello and IHU packets, and the update timer, which governs the sending of periodic route updates.

3.2.4. The Neighbour Table

The neighbour table contains the list of all neighbouring interfaces from which a Babel packet has been recently received. The neighbour table is indexed by pairs of the form (interface, address), and every neighbour table entry contains the following data:

- the local node's interface over which this neighbour is reachable;

- the address of the neighbouring interface;

- a history of recently received Multicast Hello packets from this neighbour; this can, for example, be a sequence of n bits, for some small value n, indicating which of the n hellos most recently sent by this neighbour have been received by the local node;

- a history of recently received Unicast Hello packets from this neighbour;

- the "transmission cost" value from the last IHU packet received from this neighbour, or FFFF hexadecimal (infinity) if the IHU hold timer for this neighbour has expired;

- the neighbour's expected incoming Multicast Hello sequence number, an integer modulo 2^16.

- the neighbour's expected incoming Unicast Hello sequence number, an integer modulo 2^16.

- the neighbour's outgoing Unicast Hello sequence number, an integer modulo 2^16 that is sent with each Unicast Hello TLV to this neighbour and is incremented (modulo 2^16) whenever a Unicast Hello is sent. (Note that a neighbour's outgoing Unicast Hello seqno is distinct from the interface's outgoing Multicast Hello seqno.)

There are three timers associated with each neighbour entry -- the multicast hello timer, which is initialised from the interval value carried by scheduled Multicast Hello TLVs, the unicast hello timer, which is initialised from the interval value carried by scheduled Unicast Hello TLVs, and the IHU timer, which is initialised to a small multiple of the interval carried in IHU TLVs.

Note that the neighbour table is indexed by IP addresses, not by router-ids: neighbourship is a relationship between interfaces, not between nodes. Therefore, two nodes with multiple interfaces can participate in multiple neighbourship relationships, a situation that can notably arise when wireless nodes with multiple radios are involved.

3.2.5. The Source Table

The source table is used to record feasibility distances. It is indexed by triples of the form (prefix, plen, router-id), and every source table entry contains the following data:

- the prefix (prefix, plen), where plen is the prefix length, that this entry applies to;

- the router-id of a router originating this prefix;

- a pair (seqno, metric), this source's feasibility distance.

There is one timer associated with each entry in the source table -- the source garbage-collection timer. It is initialised to a time on the order of minutes and reset as specified in Section 3.7.3.

3.2.6. The Route Table

The route table contains the routes known to this node. It is indexed by triples of the form (prefix, plen, neighbour), and every route table entry contains the following data:

- the source (prefix, plen, router-id) for which this route is advertised;

- the neighbour that advertised this route;

- the metric with which this route was advertised by the neighbour, or FFFF hexadecimal (infinity) for a recently retracted route;

- the sequence number with which this route was advertised;

- the next-hop address of this route;

- a boolean flag indicating whether this route is selected, i.e., whether it is currently being used for forwarding and is being advertised.

There is one timer associated with each route table entry -- the route expiry timer. It is initialised and reset as specified in Section 3.5.4.

Note that there are two distinct (seqno, metric) pairs associated to each route: the route's distance, which is stored in the route table, and the feasibility distance, stored in the source table and shared between all routes with the same source.

3.2.7. The Table of Pending Seqno Requests

The table of pending seqno requests contains a list of seqno requests that the local node has sent (either because they have been originated locally, or because they were forwarded) and to which no reply has been received yet. This table is indexed by triples of the form (prefix, plen, router-id), and every entry in this table contains the following data:

- the prefix, router-id, and seqno being requested;

- the neighbour, if any, on behalf of which we are forwarding this request;

- a small integer indicating the number of times that this request will be resent if it remains unsatisfied.

There is one timer associated with each pending seqno request; it governs both the resending of requests and their expiry.

3.3. Acknowledgments and acknowledgment requests

A Babel speaker may request that a neighbour receiving a given packet reply with an explicit acknowledgment within a given time. While the use of acknowledgment requests is optional, every Babel speaker MUST be able to reply to such a request.

An acknowledgment MUST be sent to a unicast destination. On the other hand, acknowledgment requests may be sent to either unicast or multicast destinations, in which case they request an acknowledgment from all of the receiving nodes.

When to request acknowledgments is a matter of local policy; the simplest strategy is to never request acknowledgments and to rely on periodic updates to ensure that any reachable routes are eventually propagated throughout the routing domain. In order to improve convergence speed and reduce the amount of control traffic, acknowledgment requests MAY be used in order to reliably send urgent updates (Section 3.7.2) and retractions (Section 3.5.5), especially when the number of neighbours on a given interface is small. Since Babel is designed to deal gracefully with packet loss on unreliable media, sending all packets with acknowledgment requests is not necessary, and NOT RECOMMENDED, as the acknowledgments cause additional traffic and may force additional Address Resolution Protocol (ARP) or Neighbour Discovery (ND) exchanges.

3.4. Neighbour Acquisition

Neighbour acquisition is the process by which a Babel node discovers the set of neighbours heard over each of its interfaces and ascertains bidirectional reachability. On unreliable media, neighbour acquisition additionally provides some statistics that may be useful for link quality computation.

Before it can exchange routing information with a neighbour, a Babel node MUST create an entry for that neighbour in the neighbour table. When to do that is implementation-specific; suitable strategies include creating an entry when any Babel packet is received, or creating an entry when a Hello TLV is parsed. Similarly, in order to conserve system resources, an implementation SHOULD discard an entry when it has been unused for long enough; suitable strategies include dropping the neighbour after a timeout, and dropping a neighbour when the associated Hello histories become empty (see Appendix A.2).

3.4.1. Reverse Reachability Detection

Every Babel node sends Hello TLVs to its neighbours to indicate that it is alive, at regular or irregular intervals. Each Hello TLV carries an increasing (modulo 2^16) sequence number and an upper bound on the time interval until the next Hello of the same type (see below). If the time interval is set to 0, then the Hello TLV does not establish a new promise: the deadline carried by the previous Hello of the same type still applies to the next Hello (if the most recent scheduled Hello of the right kind was received at time t0 and carried interval i, then the previous promise of sending another Hello before time t0 + i still holds). We say that a Hello is "scheduled" if it carries a non-zero interval, and "unscheduled" otherwise.

There are two kinds of Hellos: Multicast Hellos, which use a per-interface Hello counter (the Multicast Hello seqno), and Unicast Hellos, which use a per-neighbour counter (the Multicast Hello seqno). A Multicast Hello with a given seqno MUST be sent to all neighbours on a given interface, either by sending it to a multicast address or by sending it to one unicast address per neighbour (hence, the term "Multicast Hello" is a slight misnomer). A Unicast Hello carrying a given seqno should normally be sent to just one neighbour (over unicast), since the sequence numbers of different neighbours are not in general synchronised.

Multicast Hellos sent over multicast can be used for neighbour discovery; hence, a node SHOULD send periodic (scheduled) Multicast Hellos unless neighbour discovery is performed by means outside of the Babel protocol. A node MAY send Unicast Hellos or unscheduled Hellos of either kind for any reason, such as reducing the amount of multicast traffic or improving reliability on link technologies with poor support for link-layer multicast.

A node MAY send a scheduled Hello ahead of time. A node MAY change its scheduled Hello interval. The Hello interval MAY be decreased at any time; it MAY be increased immediately before sending a Hello TLV, but SHOULD NOT be increased at other times. (Equivalently, a node SHOULD send a scheduled Hello immediately after increasing its Hello interval.)

How to deal with received Hello TLVs and what statistics to maintain are considered local implementation matters; typically, a node will maintain some sort of history of recently received Hellos. An example of a suitable algorithm is described in Appendix A.1.

After receiving a Hello, or determining that it has missed one, the node recomputes the association’s cost (Section 3.4.3) and runs the route selection procedure (Section 3.6).

3.4.2. Bidirectional Reachability Detection

In order to establish bidirectional reachability, every node sends periodic IHU ("I Heard You") TLVs to each of its neighbours. Since IHUs carry an explicit interval value, they MAY be sent less often than Hellos in order to reduce the amount of routing traffic in dense networks; in particular, they SHOULD be sent less often than Hellos over links with little packet loss. While IHUs are conceptually unicast, they MAY be sent to a multicast address in order to avoid an ARP or Neighbour Discovery exchange and to aggregate multiple IHUs into a single packet.

In addition to the periodic IHUs, a node MAY, at any time, send an unscheduled IHU packet. It MAY also, at any time, decrease its IHU interval, and it MAY increase its IHU interval immediately before sending an IHU, but SHOULD NOT increase it at any other time.

(Equivalently, a node SHOULD send an extra IHU immediately after increasing its Hello interval.)

Every IHU TLV contains two pieces of data: the link's rxcost (reception cost) from the sender's perspective, used by the neighbour for computing link costs (Section 3.4.3), and the interval between periodic IHU packets. A node receiving an IHU sets the value of the txcost (transmission cost) maintained in the neighbour table to the value contained in the IHU, and resets the IHU timer associated to this neighbour to a small multiple of the interval value received in the IHU. When a neighbour's IHU timer expires, the neighbour's txcost is set to infinity.

After updating a neighbour's txcost, the receiving node recomputes the neighbour's cost (Section 3.4.3) and runs the route selection procedure (Section 3.6).

3.4.3. Cost Computation

A neighbourship association's link cost is computed from the values maintained in the neighbour table: the statistics kept in the neighbour table about the reception of Hellos, and the txcost computed from received IHU packets.

For every neighbour, a Babel node computes a value known as this neighbour's rxcost. This value is usually derived from the Hello history, which may be combined with other data, such as statistics maintained by the link layer. The rxcost is sent to a neighbour in each IHU.

Since nodes do not necessarily send periodic Unicast Hellos but do usually send periodic Multicast Hellos (Section 3.4.1), a node SHOULD use an algorithm that yields a finite rxcost when only Multicast Hellos are received, unless interoperability with nodes that only send Multicast Hellos is not required.

How the txcost and rxcost are combined in order to compute a link's cost is a matter of local policy; as far as Babel's correctness is concerned, only the following conditions MUST be satisfied:

o the cost is strictly positive;

o if no Hello TLVs of either kind were received recently, then the cost is infinite;

o if the txcost is infinite, then the cost is infinite.

Note that while this document does not constrain cost computation any further, not all cost computation strategies will give good results. See Appendix A.2 for examples of strategies for computing a link's cost that are known to work well in practice.

3.5. Routing Table Maintenance

Conceptually, a Babel update is a quintuple (prefix, plen, router-id, seqno, metric), where (prefix, plen) is the prefix for which a route is being advertised, router-id is the router-id of the router originating this update, seqno is a nondecreasing (modulo 2^16) integer that carries the originating router seqno, and metric is the announced metric.

Before being accepted, an update is checked against the feasibility condition (Section 3.5.1), which ensures that the route does not create a routing loop. If the feasibility condition is not satisfied, the update is either ignored or prevents the route from being selected, as described in Section 3.5.4. If the feasibility condition is satisfied, then the update cannot possibly cause a routing loop.

3.5.1. The Feasibility Condition

The feasibility condition is applied to all received updates. The feasibility condition compares the metric in the received update with the metrics of the updates previously sent by the receiving node; updates that fail the feasibility condition, and therefore have metrics large enough to cause a routing loop, are either ignored or prevent the resulting route from being selected.

A feasibility distance is a pair (seqno, metric), where seqno is an integer modulo 2^16 and metric is a positive integer. Feasibility distances are compared lexicographically, with the first component inverted: we say that a distance (seqno, metric) is strictly better than a distance (seqno', metric'), written

```
   (seqno, metric) < (seqno', metric')
```

when

```
   seqno > seqno' or (seqno = seqno' and metric < metric')
```

where sequence numbers are compared modulo 2^16.

Given a source (prefix, plen, router-id), a node's feasibility distance for this source is the minimum, according to the ordering defined above, of the distances of all the finite updates ever sent by this particular node for the prefix (prefix, plen) and the given router-id. Feasibility distances are maintained in the source table, the exact procedure is given in Section 3.7.3.

A received update is feasible when either it is a retraction (its metric is FFFF hexadecimal), or the advertised distance is strictly better, in the sense defined above, than the feasibility distance for the corresponding source. More precisely, a route advertisement carrying the quintuple (prefix, plen, router-id, seqno, metric) is feasible if one of the following conditions holds:

- metric is infinite; or
- no entry exists in the source table indexed by (prefix, plen, router-id); or
- an entry (prefix, plen, router-id, seqno', metric') exists in the source table, and either
  - seqno' < seqno or
  - seqno = seqno' and metric < metric'.

Note that the feasibility condition considers the metric advertised by the neighbour, not the route's metric; hence, a fluctuation in a neighbour's cost cannot render a selected route unfeasible. Note further that retractions (updates with infinite metric) are always feasible, since they cannot possibly cause a routing loop.

3.5.2. Metric Computation

A route's metric is computed from the metric advertised by the neighbour and the neighbour's link cost. Just like cost computation, metric computation is considered a local policy matter; as far as Babel is concerned, the function M(c, m) used for computing a metric from a locally computed link cost and the metric advertised by a neighbour MUST only satisfy the following conditions:

- if c is infinite, then M(c, m) is infinite;
- M is strictly monotonic: M(c, m) > m.

Additionally, the metric SHOULD satisfy the following condition:

- M is left-distributive: if m <= m', then M(c, m) <= M(c, m').

Note that while strict monotonicity is essential to the integrity of the network (persistent routing loops may arise if it is not satisfied), left distributivity is not: if it is not satisfied, Babel will still converge to a loop-free configuration, but might not reach a global optimum (in fact, a global optimum may not even exist).

As with cost computation, not all strategies for computing route metrics will give good results. In particular, some metrics are more likely than others to lead to routing instabilities (route flapping). In Appendix A.3, we give a number of examples of strictly monotonic, left-distributive routing metrics that are known to work well in practice.

3.5.3. Encoding of Updates

In a large network, the bulk of Babel traffic consists of route updates; hence, some care has been given to encoding them efficiently. An Update TLV itself only contains the prefix, seqno, and metric, while the next hop is derived either from the network-layer source address of the packet or from an explicit Next Hop TLV in the same packet. The router-id is derived from a separate Router-Id TLV in the same packet, which optimises the case when multiple updates are sent with the same router-id.

Additionally, a prefix of the advertised prefix can be omitted in an Update TLV, in which case it is copied from a previous Update TLV in the same packet -- this is known as address compression (Section 4.6.9).

Finally, as a special optimisation for the case when a router-id coincides with the interface-id part of an IPv6 address, the router-id can optionally be derived from the low-order bits of the advertised prefix.

The encoding of updates is described in detail in Section 4.6.

3.5.4. Route Acquisition

When a Babel node receives an update (prefix, plen, router-id, seqno, metric) from a neighbour neigh with a link cost value equal to cost, it checks whether it already has a route table entry indexed by (prefix, plen, neigh).

If no such entry exists:

- if the update is unfeasible, it MAY be ignored;

- if the metric is infinite (the update is a retraction of a route we do not know about), the update is ignored;

- otherwise, a new entry is created in the route table, indexed by (prefix, plen, neigh), with source equal to (prefix, plen, router-id), seqno equal to seqno and an advertised metric equal to the metric carried by the update.

If such an entry exists:

- if the entry is currently selected, the update is unfeasible, and the router-id of the update is equal to the router-id of the entry, then the update MAY be ignored;

- otherwise, the entry's sequence number, advertised metric, metric, and router-id are updated and, if the advertised metric is not infinite, the route's expiry timer is reset to a small multiple of the Interval value included in the update. If the update is unfeasible, then the (now unfeasible) entry MUST be immediately unselected. If the update caused the router-id of the entry to change, an update (possibly a retraction) MUST be sent in a timely manner (see Section 3.7.2).

Note that the route table may contain unfeasible routes, either because they were created by an unfeasible update or due to a metric fluctuation. Such routes are never selected, since they are not known to be loop-free; should all the feasible routes become unusable, however, the unfeasible routes can be made feasible and therefore possible to select by sending requests along them (see Section 3.8.2).

When a route's expiry timer triggers, the behaviour depends on whether the route's metric is finite. If the metric is finite, it is set to infinity and the expiry timer is reset. If the metric is already infinite, the route is flushed from the route table.

After the route table is updated, the route selection procedure (Section 3.6) is run.

3.5.5. Hold Time

When a prefix P is retracted, because all routes are unfeasible or have an infinite metric (whether due to the expiry timer or to other reasons), and a shorter prefix P' that covers P is reachable, P' cannot in general be used for routing packets destined to P without running the risk of creating a routing loop (Section 2.8).

To avoid this issue, whenever a prefix P is retracted, a route table entry with infinite metric is maintained as described in Section 3.5.4 above. As long as this entry is maintained, packets destined to an address within P MUST NOT be forwarded by following a route for a shorter prefix. This entry is removed as soon as a finite-metric update for prefix P is received and the resulting route selected. If no such update is forthcoming, the infinite metric entry SHOULD be maintained at least until it is guaranteed that no neighbour has selected the current node as next-hop for prefix P. This can be achieved by either:

- waiting until the route's expiry timer has expired (Section 3.5.4);

- sending a retraction with an acknowledgment request (Section 3.3) to every reachable neighbour that has not explicitly retracted prefix P and waiting for all acknowledgments.

The former option is simpler and ensures that at that point, any routes for prefix P pointing at the current node have expired. However, since the expiry time can be as high as a few minutes, doing that prevents automatic aggregation by creating spurious black-holes for aggregated routes. The latter option is RECOMMENDED as it dramatically reduces the time for which a prefix is unreachable in the presence of aggregated routes.

3.6. Route Selection

Route selection is the process by which a single route for a given prefix is selected to be used for forwarding packets and to be re-advertised to a node's neighbours.

Babel is designed to allow flexible route selection policies. As far as the protocol's correctness is concerned, the route selection policy MUST only satisfy the following properties:

- a route with infinite metric (a retracted route) is never selected;

- an unfeasible route is never selected.

Note, however, that Babel does not naturally guarantee the stability of routing, and configuring conflicting route selection policies on different routers may lead to persistent route oscillation.

Route selection is a difficult problem, since a good route selection policy needs to take into account multiple mutually contradictory criteria; in roughly decreasing order of importance, these are:

- routes with a small metric should be preferred to routes with a large metric;

- switching router-ids should be avoided;

- routes through stable neighbours should be preferred to routes through unstable ones;

- stable routes should be preferred to unstable ones;

- switching next hops should be avoided.

A simple but useful strategy is to choose the feasible route with the smallest metric, with a small amount of hysteresis applied to avoid switching router-ids too often.

After the route selection procedure is run, triggered updates (Section 3.7.2) and requests (Section 3.8.2) are sent.

3.7. Sending Updates

A Babel speaker advertises to its neighbours its set of selected routes. Normally, this is done by sending one or more multicast packets containing Update TLVs on all of its connected interfaces; however, on link technologies where multicast is significantly more expensive than unicast, a node MAY choose to send multiple copies of updates in unicast packets, especially when the number of neighbours is small.

Additionally, in order to ensure that any black-holes are reliably cleared in a timely manner, a Babel node sends retractions (updates with an infinite metric) for any recently retracted prefixes.

If an update is for a route injected into the Babel domain by the local node (e.g., it carries the address of a local interface, the prefix of a directly attached network, or a prefix redistributed from a different routing protocol), the router-id is set to the local node’s router-id, the metric is set to some arbitrary finite value (typically 0), and the seqno is set to the local router’s sequence number.

If an update is for a route learned from another Babel speaker, the router-id and sequence number are copied from the route table entry, and the metric is computed as specified in Section 3.5.2.

3.7.1. Periodic Updates

Every Babel speaker periodically advertises all of its selected routes on all of its interfaces, including any recently retracted routes. Since Babel doesn’t suffer from routing loops (there is no "counting to infinity") and relies heavily on triggered updates (Section 3.7.2), this full dump only needs to happen infrequently.

3.7.2. Triggered Updates

In addition to periodic routing updates, a Babel speaker sends unscheduled, or triggered, updates in order to inform its neighbours of a significant change in the network topology.

A change of router-id for the selected route to a given prefix may be indicative of a routing loop in formation; hence, a node MUST send a triggered update in a timely manner whenever it changes the selected router-id for a given destination. Additionally, it SHOULD make a reasonable attempt at ensuring that all reachable neighbours receive this update.

There are two strategies for ensuring that. If the number of neighbours is small, then it is reasonable to send the update together with an acknowledgment request; the update is resent until all neighbours have acknowledged the packet, up to some number of times. If the number of neighbours is large, however, requesting acknowledgments from all of them might cause a non-negligible amount of network traffic; in that case, it may be preferable to simply repeat the update some reasonable number of times (say, 5 for wireless and 2 for wired links).

A route retraction is somewhat less worrying: if the route retraction doesn’t reach all neighbours, a black-hole might be created, which, unlike a routing loop, does not endanger the integrity of the network. When a route is retracted, a node SHOULD send a triggered update and SHOULD make a reasonable attempt at ensuring that all neighbours receive this retraction.

Finally, a node MAY send a triggered update when the metric for a given prefix changes in a significant manner, due to a received update, because a link’s cost has changed, or because a different next hop has been selected. A node SHOULD NOT send triggered updates for other reasons, such as when there is a minor fluctuation in a route’s metric, when the selected next hop changes, or to propagate a new sequence number (except to satisfy a request, as specified in Section 3.8).

3.7.3. Maintaining Feasibility Distances

Before sending an update (prefix, plen, router-id, seqno, metric) with finite metric (i.e., not a route retraction), a Babel node updates the feasibility distance maintained in the source table. This is done as follows.

If no entry indexed by (prefix, plen, router-id) exists in the source table, then one is created with value (prefix, plen, router-id, seqno, metric).

If an entry (prefix, plen, router-id, seqno’, metric’) exists, then it is updated as follows:

- if seqno > seqno’, then seqno’ := seqno, metric’ := metric;

- if seqno = seqno’ and metric’ > metric, then metric’ := metric;

- otherwise, nothing needs to be done.

The garbage-collection timer for the entry is then reset. Note that the feasibility distance is not updated and the garbage-collection timer is not reset when a retraction (an update with infinite metric) is sent.

When the garbage-collection timer expires, the entry is removed from the source table.

3.7.4. Split Horizon

When running over a transitive, symmetric link technology, e.g., a point-to-point link or a wired LAN technology such as Ethernet, a Babel node SHOULD use an optimisation known as split horizon. When split horizon is used on a given interface, a routing update for prefix P is not sent on the particular interface over which the selected route towards prefix P was learnt.

Split horizon SHOULD NOT be applied to an interface unless the interface is known to be symmetric and transitive; in particular, split horizon is not applicable to decentralised wireless link technologies (e.g., IEEE 802.11 in ad hoc mode) when routing updates are sent over multicast.

3.8. Explicit Requests

In normal operation, a node’s route table is populated by the regular and triggered updates sent by its neighbours. Under some circumstances, however, a node sends explicit requests in order to cause a resynchronisation with the source after a mobility event or to prevent a route from spuriously expiring.

The Babel protocol provides two kinds of explicit requests: route requests, which simply request an update for a given prefix, and seqno requests, which request an update for a given prefix with a specific sequence number. The former are never forwarded; the latter are forwarded if they cannot be satisfied by the receiver.

3.8.1. Handling Requests

Upon receiving a request, a node either forwards the request or sends an update in reply to the request, as described in the following sections. If this causes an update to be sent, the update is either sent to a multicast address on the interface on which the request was received, or to the unicast address of the neighbour that sent the request.

The exact behaviour is different for route requests and seqno requests.

3.8.1.1. Route Requests

When a node receives a route request for a given prefix, it checks its route table for a selected route to this exact prefix. If such a route exists, it MUST send an update (over unicast or over multicast); if such a route does not exist, it MUST send a retraction for that prefix.

When a node receives a wildcard route request, it SHOULD send a full route table dump. Full route dumps MAY be rate-limited, especially if they are sent over multicast.

3.8.1.2. Seqno Requests

When a node receives a seqno request for a given router-id and sequence number, it checks whether its route table contains a selected entry for that prefix. If a selected route for the given prefix exists, it has finite metric, and either the router-ids are different or the router-ids are equal and the entry’s sequence number is no smaller (modulo 2^16) than the requested sequence number, the node MUST send an update for the given prefix. If the router-ids match but the requested seqno is larger (modulo 2^16) than the route entry’s, the node compares the router-id against its own router-id. If the router-id is its own, then it increases its sequence number by 1 (modulo 2^16) and sends an update. A node MUST NOT increase its sequence number by more than 1 in response to a seqno request.

Otherwise, if the requested router-id is not its own, the received request’s hop count is 2 or more, and the node is advertising the prefix to its neighbours, the node selects a neighbour to forward the request to as follows:

- if the node has one or more feasible routes toward the requested prefix with a next hop that is not the requesting node, then the node MUST forward the request to the next hop of one such route;

- otherwise, if the node has one or more (not necessarily feasible) routes to the requested prefix with a next hop that is not the requesting node, then the node SHOULD forward the request to the next hop of one such route.

In order to actually forward the request, the node decrements the hop count and sends the request in a unicast packet destined to the selected neighbour.

A node SHOULD maintain a list of recently forwarded seqno requests and forward the reply (an update with a seqno sufficiently large to satisfy the request) in a timely manner. A node SHOULD compare every incoming seqno request against its list of recently forwarded seqno requests and avoid forwarding it if it is redundant (i.e., if it has recently sent a request with the same prefix, router-id and a seqno that is not smaller modulo 2^16).

Since the request-forwarding mechanism does not necessarily obey the feasibility condition, it may get caught in routing loops; hence, requests carry a hop count to limit the time during which they remain in the network. However, since requests are only ever forwarded as unicast packets, the initial hop count need not be kept particularly low, and performing an expanding horizon search is not necessary. A single request MUST NOT be duplicated: it MUST NOT be forwarded to a multicast address, and it MUST NOT be forwarded to multiple neighbours. However, if a seqno request is resent by its originator, the subsequent copies MAY be forwarded to a different neighbour than the initial one.

3.8.2. Sending Requests

A Babel node MAY send a route or seqno request at any time, to a multicast or a unicast address; there is only one case when originating requests is required (Section 3.8.2.1).

3.8.2.1. Avoiding Starvation

When a route is retracted or expires, a Babel node usually switches to another feasible route for the same prefix. It may be the case, however, that no such routes are available.

A node that has lost all feasible routes to a given destination but still has unexpired unfeasible routes to that destination MUST send a seqno request; if it doesn't have any such routes, it MAY still send a seqno request. The router-id of the request is set to the router-id of the route that it has just lost, and the requested seqno is the value contained in the source table plus 1.

If the node has any (unfeasible) routes to the requested destination, then it MUST send the request to at least one of the next-hop neighbours that advertised these routes, and SHOULD send it to all of them; in any case, it MAY send the request to any other neighbours, whether they advertise a route to the requested destination or not. A simple implementation strategy is therefore to unconditionally multicast the request over all interfaces.

Similar requests will be sent by other nodes that are affected by the route's loss. If the network is still connected, and assuming no packet loss, then at least one of these requests will be forwarded to the source, resulting in a route being advertised with a new sequence number. (Due to duplicate suppression, only a small number of such requests will actually reach the source.)

In order to compensate for packet loss, a node SHOULD repeat such a request a small number of times if no route becomes feasible within a short time. In the presence of heavy packet loss, however, all such requests might be lost; in that case, the mechanism in the next section will eventually ensure that a new seqno is received.

3.8.2.2. Dealing with Unfeasible Updates

When a route's metric increases, a node might receive an unfeasible update for a route that it has currently selected. As specified in Section 3.5.1, the receiving node will either ignore the update or unselect the route.

In order to keep routes from spuriously expiring because they have become unfeasible, a node SHOULD send a unicast seqno request when it receives an unfeasible update for a route that is currently selected. The requested sequence number is computed from the source table as in Section 3.8.2.1 above.

Additionally, since metric computation does not necessarily coincide with the delay in propagating updates, a node might receive an unfeasible update from a currently unselected neighbour that is preferable to the currently selected route (e.g., because it has a much smaller metric); in that case, the node SHOULD send a unicast seqno request to the neighbour that advertised the preferable update.

3.8.2.3. Preventing Routes from Expiring

In normal operation, a route's expiry timer never triggers: since a route's hold time is computed from an explicit interval included in Update TLVs, a new update (possibly a retraction) should arrive in time to prevent a route from expiring.

In the presence of packet loss, however, it may be the case that no update is successfully received for an extended period of time, causing a route to expire. In order to avoid such spurious expiry, shortly before a selected route expires, a Babel node SHOULD send a unicast route request to the neighbour that advertised this route; since nodes always send either updates or retractions in response to non-wildcard route requests (Section 3.8.1.1), this will usually result in the route being either refreshed or retracted.

3.8.2.4. Acquiring New Neighbours

In order to speed up convergence after a mobility event, a node MAY send a unicast wildcard request after acquiring a new neighbour. Additionally, a node MAY send a small number of multicast wildcard requests shortly after booting. Note however that doing that carelessly can cause serious congestion when a whole network is rebooted, especially on link layers with high per-packet overhead (e.g., IEEE 802.11).

4. Protocol Encoding

A Babel packet is sent as the body of a UDP datagram, with network-layer hop count set to 1, destined to a well-known multicast address or to a unicast address, over IPv4 or IPv6; in the case of IPv6, these addresses are link-local. Both the source and destination UDP port are set to a well-known port number. A Babel packet MUST be silently ignored unless its source address is either a link-local IPv6 address or an IPv4 address belonging to the local network, and its source port is the well-known Babel port. It MAY be silently ignored if its destination address is a global IPv6 address.

In order to minimise the number of packets being sent while avoiding lower-layer fragmentation, a Babel node SHOULD attempt to maximise the size of the packets it sends, up to the outgoing interface's MTU adjusted for lower-layer headers (28 octets for UDP over IPv4, 48 octets for UDP over IPv6). It MUST NOT send packets larger than the attached interface's MTU adjusted for lower-layer headers or 512 octets, whichever is larger, but not exceeding 2^16 - 1 adjusted for lower-layer headers. Every Babel speaker MUST be able to receive packets that are as large as any attached interface's MTU adjusted for lower-layer headers or 512 octets, whichever is larger. Babel packets MUST NOT be sent in IPv6 Jumbograms.

In order to avoid global synchronisation of a Babel network and to aggregate multiple TLVs into large packets, a Babel node SHOULD buffer every TLV and delay sending a packet by a small, randomly chosen delay [JITTER]. In order to allow accurate computation of packet loss rates, this delay MUST NOT be larger than half the advertised Hello interval.

4.1. Data Types

4.1.1. Interval

Relative times are carried as 16-bit values specifying a number of centiseconds (hundredths of a second). This allows times up to roughly 11 minutes with a granularity of 10ms, which should cover all reasonable applications of Babel.

4.1.2. Router-Id

A router-id is an arbitrary 8-octet value. A router-id MUST NOT consist of either all zeroes or all ones.

4.1.3. Address

Since the bulk of the protocol is taken by addresses, multiple ways of encoding addresses are defined. Additionally, a common subnet prefix may be omitted when multiple addresses are sent in a single packet -- this is known as address compression (Section 4.6.9).

Address encodings:

- AE 0: wildcard address. The value is 0 octets long.

- AE 1: IPv4 address. Compression is allowed. 4 octets or less.

- AE 2: IPv6 address. Compression is allowed. 16 octets or less.

- AE 3: link-local IPv6 address. Compression is not allowed. The value is 8 octets long, a prefix of fe80::/64 is implied.

The address family associated to an address encoding is either IPv4 or IPv6; it is undefined for AE 0, IPv4 for AE 1, and IPv6 for AEs 2 and 3.

4.1.4. Prefixes

A network prefix is encoded just like a network address, but it is stored in the smallest number of octets that are enough to hold the significant bits (up to the prefix length).

4.2. Packet Format

A Babel packet consists of a 4-octet header, followed by a sequence of TLVs (the packet body), optionally followed by a second sequence of TLVs (the packet trailer).

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Magic     |    Version    |        Body length            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Packet Body ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
|         Packet Trailer...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
```

Fields :

Magic   The arbitrary but carefully chosen value 42 (decimal); packets with a first octet different from 42 MUST be silently ignored.

Version   This document specifies version 2 of the Babel protocol. Packets with a second octet different from 2 MUST be silently ignored.

Body length The length in octets of the body following the packet header (excluding the Magic, Version and Body length fields, and excluding the packet trailer).

Packet Body The packet body; a sequence of TLVs.

Packet Trailer The packet trailer; another sequence of TLVs.

The packet body and trailer are both sequences of TLVs. The packet body is the normal place to store TLVs; the packet trailer only contains specialised TLVs that do not need to be protected by cryptographic security mechanisms. When parsing the trailer, the receiver MUST ignore any TLV unless its definition explicitly states that it is allowed to appear there. Among the TLVs defined in this document, only Pad1 and PadN are allowed in the trailer; since these TLVs are ignored in any case, an implementation MAY silently ignore the packet trailer without even parsing it, unless it implements at least one extension that defines TLVs that are allowed to appear in the trailer.

4.3. TLV Format

With the exception of Pad1, all TLVs have the following structure:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Type      |    Length     |     Payload...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
```

Fields :

Type   The type of the TLV.

Length   The length of the body, exclusive of the Type and Length fields. If the body is longer than the expected length of a given type of TLV, any extra data MUST be silently ignored.

Payload   The TLV payload, which consists of a body and, for selected TLV types, an optional list of sub-TLVs.

TLVs with an unknown type value MUST be silently ignored.

4.4. Sub-TLV Format

Every TLV carries an explicit length in its header; however, most TLVs are self-terminating, in the sense that it is possible to determine the length of the body without reference to the explicit Length field. If a TLV has a self-terminating format, then it MAY allow a sequence of sub-TLVs to follow the body.

Sub-TLVs have the same structure as TLVs. With the exception of PAD1, all TLVs have the following structure:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Type      |    Length     |     Body...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
```

Fields :

Type   The type of the sub-TLV.

Length The length of the body, in octets, exclusive of the Type and Length fields.

Body The sub-TLV body, the interpretation of which depends on both the type of the sub-TLV and the type of the TLV within which it is embedded.

The most-significant bit of the sub-TLV, called the mandatory bit, indicates how to handle unknown sub-TLVs. If the mandatory bit is not set, then an unknown sub-TLV MUST be silently ignored, and the rest of the TLV processed normally. If the mandatory bit is set, then the whole enclosing TLV MUST be silently ignored (except for updating the parser state by a Router-Id, Next-Hop or Update TLV, see Section 4.6.7, Section 4.6.8, and Section 4.6.9).

4.5. Parser state

Babel uses a stateful parser: a TLV may refer to data from a previous TLV. The parser state consists of the following pieces of data:

- for each address encoding that allows compression, the current default prefix; this is undefined at the start of the packet, and is updated by each Update TLV with the Prefix flag set (Section 4.6.9);

- for each address family (IPv4 or IPv6), the current next-hop; this is the source address of the enclosing packet for the matching address family at the start of a packet, and is updated by each Next-Hop TLV (Section 4.6.8);

- the current router-id; this is undefined at the start of the packet, and is updated by each Router-ID TLV (Section 4.6.7) and by each Update TLV with Router-Id flag set.

Since the parser state is separate from the bulk of Babel's state, and since for correct parsing it must be identical across implementations, it is updated before checking for mandatory TLVs: parsing a TLV MUST update the parser state even if the TLV is otherwise ignored due to an unknown mandatory sub-TLV.

None of the TLVs that modify the parser state are allowed in the packet trailer; hence, an implementation may choose to use a dedicated stateless parser to parse the packet trailer.

4.6. Details of Specific TLVs

4.6.1. Pad1

```
    0
    0 1 2 3 4 5 6 7
   +-+-+-+-+-+-+-+-+
   |   Type = 0    |
   +-+-+-+-+-+-+-+-+
```

Fields :

Type   Set to 0 to indicate a Pad1 TLV.

This TLV is silently ignored on reception. It is allowed in the packet trailer.

4.6.2. PadN

```
    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |    Type = 1   |    Length     |      MBZ...
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
```

Fields :

Type   Set to 1 to indicate a PadN TLV.

Length   The length of the body, exclusive of the Type and Length fields.

MBZ   Set to 0 on transmission.

This TLV is silently ignored on reception. It is allowed in the packet trailer.

4.6.3. Acknowledgment Request

```
    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |    Type = 2   |    Length     |          Reserved             |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |             Nonce             |          Interval             |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

This TLV requests that the receiver send an Acknowledgment TLV within the number of centiseconds specified by the Interval field.

Fields :

Type   Set to 2 to indicate an Acknowledgment Request TLV.

Length   The length of the body, exclusive of the Type and Length fields.

Reserved Sent as 0 and MUST be ignored on reception.

Nonce   An arbitrary value that will be echoed in the receiver's Acknowledgment TLV.

Interval A time interval in centiseconds after which the sender will assume that this packet has been lost. This MUST NOT be 0. The receiver MUST send an Acknowledgment TLV before this time has elapsed (with a margin allowing for propagation time).

This TLV is self-terminating, and allows sub-TLVs.

4.6.4. Acknowledgment

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Type = 3   |    Length     |            Nonce              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

This TLV is sent by a node upon receiving an Acknowledgment Request.

Fields :

Type   Set to 3 to indicate an Acknowledgment TLV.

Length   The length of the body, exclusive of the Type and Length fields.

Nonce   Set to the Nonce value of the Acknowledgment Request that prompted this Acknowledgment.

Since nonce values are not globally unique, this TLV MUST be sent to a unicast address.

This TLV is self-terminating, and allows sub-TLVs.

4.6.5. Hello

```
    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |    Type = 4   |    Length     |             Flags             |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |            Seqno              |            Interval           |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

This TLV is used for neighbour discovery and for determining a neighbour’s reception cost.

Fields :

Type   Set to 4 to indicate a Hello TLV.

Length   The length of the body, exclusive of the Type and Length fields.

Flags   The individual bits of this field specify special handling of this TLV (see below).

Seqno   If the Unicast flag is set, this is the value of the sending node’s outgoing Unicast Hello seqno for this neighbour. Otherwise, it is the sending node’s outgoing Multicast Hello seqno for this interface.

Interval If non-zero, this is an upper bound, expressed in centiseconds, on the time after which the sending node will send a new scheduled Hello TLV with the same setting of the Unicast flag. If this is 0, then this Hello represents an unscheduled Hello, and doesn’t carry any new information about times at which Hellos are sent.

The Flags field is interpreted as follows:

```
    0                   1
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |U|x|x|x|x|x|x|x|x|x|x|x|x|x|x|x|
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- U (Unicast) flag (8000 hexadecimal): if set, then this Hello represents a Unicast Hello, otherwise it represents a Multicast Hello;

- X: all other bits MUST be sent as 0 and silently ignored on reception.

Every time a Hello is sent, the corresponding seqno counter MUST be incremented. Since there is a single seqno counter for all the Multicast Hellos sent by a given node over a given interface, if the Unicast flag is not set, this TLV MUST be sent to all neighbors on this link, which can be achieved by sending to a multicast destination, or by sending multiple packets to the unicast addresses of all reachable neighbours. Conversely, if the Unicast flag is set, this TLV MUST be sent to a single neighbour, which can achieved by sending to a unicast destination. In order to avoid large discontinuities in link quality, multiple Hello TLVs SHOULD NOT be sent in the same packet.

This TLV is self-terminating, and allows sub-TLVs.

4.6.6. IHU

```
    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |    Type = 5   |    Length     |       AE      |    Reserved   |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |            Rxcost             |           Interval            |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |       Address...
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-
```

An IHU ("I Heard You") TLV is used for confirming bidirectional reachability and carrying a link’s transmission cost.

Fields :

Type   Set to 5 to indicate an IHU TLV.

Length   The length of the body, exclusive of the Type and Length fields.

AE   The encoding of the Address field. This should be 1 or 3 in most cases. As an optimisation, it MAY be 0 if the TLV is sent to a unicast address, if the association is over a point-to-point link, or when bidirectional reachability is ascertained by means outside of the Babel protocol.

Reserved Sent as 0 and MUST be ignored on reception.

Rxcost   The rxcost according to the sending node of the interface whose address is specified in the Address field. The value FFFF hexadecimal (infinity) indicates that this interface is unreachable.

Interval An upper bound, expressed in centiseconds, on the time after which the sending node will send a new IHU; this MUST NOT be 0. The receiving node will use this value in order to compute a hold time for this symmetric association.

Address   The address of the destination node, in the format specified by the AE field. Address compression is not allowed.

Conceptually, an IHU is destined to a single neighbour. However, IHU TLVs contain an explicit destination address, and MAY be sent to a multicast address, as this allows aggregation of IHUs destined to distinct neighbours into a single packet and avoids the need for an ARP or Neighbour Discovery exchange when a neighbour is not being used for data traffic.

IHU TLVs with an unknown value in the AE field MUST be silently ignored.

This TLV is self-terminating, and allows sub-TLVs.

4.6.7. Router-Id

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Type = 6   |    Length     |           Reserved            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                           Router-Id                           +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

A Router-Id TLV establishes a router-id that is implied by subsequent Update TLVs. This TLV sets the router-id even if it is otherwise ignored due to an unknown mandatory sub-TLV.

Fields :

Type   Set to 6 to indicate a Router-Id TLV.

Length   The length of the body, exclusive of the Type and Length fields.

Reserved Sent as 0 and MUST be ignored on reception.

Router-Id The router-id for routes advertised in subsequent Update TLVs. This MUST NOT consist of all zeroes or all ones.

This TLV is self-terminating, and allows sub-TLVs.

4.6.8. Next Hop

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Type = 7   |     Length    |       AE      |   Reserved    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       Next hop...
+-+-+-+-+-+-+-+-+-+-+-+-+-
```

A Next Hop TLV establishes a next-hop address for a given address family (IPv4 or IPv6) that is implied in subsequent Update TLVs. This TLV sets up the next-hop for subsequent Update TLVs even if it is otherwise ignored due to an unknown mandatory sub-TLV.

Fields :

Type Set to 7 to indicate a Next Hop TLV.

Length The length of the body, exclusive of the Type and Length fields.

AE The encoding of the Address field. This SHOULD be 1 (IPv4) or 3 (link-local IPv6), and MUST NOT be 0.

Reserved Sent as 0 and MUST be ignored on reception.

Next hop The next-hop address advertised by subsequent Update TLVs, for this address family.

When the address family matches the network-layer protocol that this packet is transported over, a Next Hop TLV is not needed: in the absence of a Next Hop TLV in a given address family, the next hop address is taken to be the source address of the packet.

Next Hop TLVs with an unknown value for the AE field MUST be silently ignored.

This TLV is self-terminating, and allows sub-TLVs.

4.6.9. Update

```
    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |    Type = 8   |     Length    |       AE      |     Flags     |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |     Plen      |    Omitted    |            Interval           |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |             Seqno             |            Metric             |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |      Prefix...
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-
```

An Update TLV advertises or retracts a route. As an optimisation, it can optionally have the side effect of establishing a new implied router-id and a new default prefix.

Fields :

Type — Set to 8 to indicate an Update TLV.

Length — The length of the body, exclusive of the Type and Length fields.

AE — The encoding of the Prefix field.

Flags — The individual bits of this field specify special handling of this TLV (see below).

Plen — The length of the advertised prefix.

Omitted — The number of octets that have been omitted at the beginning of the advertised prefix and that should be taken from a preceding Update TLV in the same address family with the Prefix flag set.

Interval — An upper bound, expressed in centiseconds, on the time after which the sending node will send a new update for this prefix. This MUST NOT be 0. The receiving node will use this value to compute a hold time for the route table entry. The value FFFF hexadecimal (infinity) expresses that this announcement will not be repeated unless a request is received (Section 3.8.2.3).

Seqno — The originator's sequence number for this update.

Metric The sender's metric for this route. The value FFFF hexadecimal (infinity) means that this is a route retraction.

Prefix The prefix being advertised. This field's size is (Plen/8 - Omitted) rounded upwards.

The Flags field is interpreted as follows:

```
 0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+
|P|R|X|X|X|X|X|X|
+-+-+-+-+-+-+-+-+
```

o P (Prefix) flag (80 hexadecimal): if set, then this Update establishes a new default prefix for subsequent Update TLVs with a matching address encoding within the same packet, even if this TLV is otherwise ignored due to an unknown mandatory sub-TLV;

o R (Router-Id) flag (40 hexadecimal): if set, then this TLV establishes a new default router-id for this TLV and subsequent Update TLVs in the same packet, even if this TLV is otherwise ignored due to an unknown mandatory sub-TLV. This router-id is computed from the first address of the advertised prefix as follows:

   * if the length of the address is 8 octets or more, then the new router-id is taken from the 8 last octets of the address;

   * if the length of the address is smaller than 8 octets, then the new router-id consists of the required number of zero octets followed by the address, i.e., the address is stored on the right of the router-id. For example, for an IPv4 address, the router-id consists of 4 octets of zeroes followed by the IPv4 address.

o X: all other bits MUST be sent as 0 and silently ignored on reception.

The prefix being advertised by an Update TLV is computed as follows:

o the first Omitted octets of the prefix are taken from the previous Update TLV with the Prefix flag set and the same address encoding, even if it was ignored due to an unknown mandatory sub-TLV;

o the next (Plen/8 - Omitted) rounded upwards octets are taken from the Prefix field;

- the remaining octets are set to 0. If AE is 3 (link-local IPv6), Omitted MUST be 0)

If the Metric field is finite, the router-id of the originating node for this announcement is taken from the prefix advertised by this Update if the Router-Id flag is set, computed as described above. Otherwise, it is taken either from the preceding Router-Id packet, or the preceding Update packet with the Router-Id flag set, whichever comes last, even if that TLV is otherwise ignored due to an unknown mandatory sub-TLV.

The next-hop address for this update is taken from the last preceding Next Hop TLV with a matching address family (IPv4 or IPv6) in the same packet even if it was otherwise ignored due to an unknown mandatory sub-TLV; if no such TLV exists, it is taken from the network-layer source address of this packet.

If the metric field is FFFF hexadecimal, this TLV specifies a retraction. In that case, the router-id, next-hop and seqno are not used. AE MAY then be 0, in which case this Update retracts all of the routes previously advertised by the sending interface. If the metric is finite, AE MUST NOT be 0. If the metric is infinite and AE is 0, Plen and Omitted MUST both be 0.

Update TLVs with an unknown value in the AE field MUST be silently ignored.

This TLV is self-terminating, and allows sub-TLVs.

4.6.10. Route Request

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Type = 9   |    Length     |      AE       |     Plen      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Prefix...
+-+-+-+-+-+-+-+-+-+-+-+-+-
```

A Route Request TLV prompts the receiver to send an update for a given prefix, or a full route table dump.

Fields :

Type   Set to 9 to indicate a Route Request TLV.

Length   The length of the body, exclusive of the Type and Length fields.

AE The encoding of the Prefix field. The value 0 specifies that this is a request for a full route table dump (a wildcard request).

Plen The length of the requested prefix.

Prefix The prefix being requested. This field’s size is Plen/8 rounded upwards.

A Request TLV prompts the receiver to send an update message (possibly a retraction) for the prefix specified by the AE, Plen, and Prefix fields, or a full dump of its route table if AE is 0 (in which case Plen MUST be 0 and Prefix is of length 0).

This TLV is self-terminating, and allows sub-TLVs.

4.6.11. Seqno Request

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Type = 10  |    Length     |      AE       |    Plen       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             Seqno             |  Hop Count    |   Reserved    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                           Router-Id                           +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Prefix...
+-+-+-+-+-+-+-+-+-+-+-+
```

A Seqno Request TLV prompts the receiver to send an Update for a given prefix with a given sequence number, or to forward the request further if it cannot be satisfied locally.

Fields :

Type Set to 10 to indicate a Seqno Request message.

Length The length of the body, exclusive of the Type and Length fields.

AE The encoding of the Prefix field. This MUST NOT be 0.

Plen The length of the requested prefix.

Seqno The sequence number that is being requested.

Hop Count The maximum number of times that this TLV may be forwarded, plus 1. This MUST NOT be 0.

Reserved Sent as 0 and MUST be ignored on reception.

Router Id The Router-Id that is being requested. This MUST NOT consist of all zeroes or all ones.

Prefix   The prefix being requested. This field's size is Plen/8 rounded upwards.

A Seqno Request TLV prompts the receiving node to send a finite-metric Update for the prefix specified by the AE, Plen, and Prefix fields, with either a router-id different from what is specified by the Router-Id field, or a Seqno no less (modulo 2^16) than what is specified by the Seqno field. If this request cannot be satisfied locally, then it is forwarded according to the rules set out in Section 3.8.1.2.

While a Seqno Request MAY be sent to a multicast address, it MUST NOT be forwarded to a multicast address and MUST NOT be forwarded to more than one neighbour. A request MUST NOT be forwarded if its Hop Count field is 1.

This TLV is self-terminating, and allows sub-TLVs.

4.7. Details of specific sub-TLVs

4.7.1. Pad1

```
 0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+
|   Type = 0    |
+-+-+-+-+-+-+-+-+
```

Fields :

Type   Set to 0 to indicate a Pad1 sub-TLV.

This sub-TLV is silently ignored on reception. It is allowed within any TLV that allows sub-TLVs.

4.7.2. PadN

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Type = 1    |    Length     |      MBZ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
```

Fields :

Type Set to 1 to indicate a PadN sub-TLV.

Length The length of the body, in octets, exclusive of the Type and Length fields.

MBZ Set to 0 on transmission.

This sub-TLV is silently ignored on reception. It is allowed within any TLV that allows sub-TLVs.

5. IANA Considerations

IANA has registered the UDP port number 6696, called "babel", for use by the Babel protocol.

IANA has registered the IPv6 multicast group ff02::1:6 and the IPv4 multicast group 224.0.0.111 for use by the Babel protocol.

IANA has created a registry called "Babel TLV Types". The values in this registry are not changed by this specification.

IANA has created a registry called "Babel sub-TLV Types". Due to the addition of a Mandatory bit to the Babel protocol, the values in the "Babel sub-TLV Types" registry are amended as follows:

Type	Name	Reference
0	Pad1	this document
1	PadN	this document
112-126	Reserved for Experimental Use	this document
127	Reserved for expansion of the type space	this document
240-254	Reserved for Experimental Use	this document
255	Reserved for expansion of the type space	this document

Existing assignments in the "Babel sub-TLV Types" registry in the range 2 to 111 are not changed by this specification. The values 224 through 239, previously reserved for Experimental Use, are now unassigned.

IANA has created a registry called "Babel Flags Values". IANA is instructed to rename this registry to "Babel Update Flags Values", with its contents unchanged.

IANA is instructed to create a new registry called "Babel Hello Flags Values". The allocation policy for this registry is Specification Required [RFC8126]. The initial values in this registry are as follows:

Bit	Name	Reference
0	Unicast	this document
1-15	Unassigned	

IANA is instructed to replace all references to RFCs 6126 and 7557 in all of the registries mentioned above by references to this document.

6. Security Considerations

As defined in this document, Babel is a completely insecure protocol. Any attacker can misdirect data traffic by advertising routes with a low metric or a high seqno. This issue can be solved either by a lower-layer security mechanism (e.g., link-layer security or IPsec), or by deploying a suitable authentication mechanism within Babel itself. There are currently two such mechanisms: Babel over DTLS [BABEL-DTLS] and HMAC-based authentication [BABEL-HMAC]. Both mechanisms ensure integrity of messages and prevent message replay, but only DTLS supports asymmetric keying and message confidentiality. HMAC is simpler and does not depend on DTLS, and therefore its use is RECOMMENDED whenever both mechanisms are applicable.

The information that a Babel node announces to the whole routing domain is often sufficient to determine a mobile node's physical location with reasonable precision. The privacy issues that this causes can be mitigated somewhat by using randomly chosen router-ids and randomly chosen IP addresses, and changing them periodically.

When carried over IPv6, Babel packets are ignored unless they are sent from a link-local IPv6 address; since routers don't forward link-local IPv6 packets, this provides protection against spoofed Babel packets being sent from the global Internet. No such natural protection exists when Babel packets are carried over IPv4.

7. Acknowledgments

A number of people have contributed text and ideas to this specification. The authors are particularly indebted to Matthieu Boutier, Gwendoline Chouasne, Margaret Cullen, Donald Eastlake and Toke Hoiland-Jorgensen. Earlier versions of this document greatly benefited from the input of Joel Halpern. The address compression technique was inspired by [PACKETBB].

8. References

8.1. Normative References

[BABEL-DTLS]
Decimo, A., Schinazi, D., and J. Chroboczek, "Babel Routing Protocol over Datagram Transport Layer Security", Internet Draft draft-ietf-babel-dtls-01, October 2018.

[BABEL-HMAC]
Do, C., Kolodziejak, W., and J. Chroboczek, "HMAC authentication for the Babel routing protocol", Internet Draft draft-ietf-babel-hmac-01, November 2018.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, June 2017.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017.

8.2. Informative References

[DSDV] Perkins, C. and P. Bhagwat, "Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers", ACM SIGCOMM'94 Conference on Communications Architectures, Protocols and Applications 234-244, 1994.

[DUAL] Garcia Luna Aceves, J., "Loop-Free Routing Using Diffusing Computations", IEEE/ACM Transactions on Networking 1:1, February 1993.

[EIGRP] Albrightson, B., Garcia Luna Aceves, J., and J. Boyle, "EIGRP -- a Fast Routing Protocol Based on Distance Vectors", Proc. Interop 94, 1994.

[ETX] De Couto, D., Aguayo, D., Bicket, J., and R. Morris, "A high-throughput path metric for multi-hop wireless networks", Proc. MobiCom 2003, 2003.

[IS-IS] "Information technology -- Telecommunications and information exchange between systems -- Intermediate System to Intermediate System intra-domain routeing information exchange protocol for use in conjunction with the protocol for providing the connectionless-mode network service (ISO 8473)", ISO/IEC 10589:2002, 2002.

[JITTER] Floyd, S. and V. Jacobson, "The synchronization of periodic routing messages", IEEE/ACM Transactions on Networking 2, 2, 122-136, April 1994.

[OSPF] Moy, J., "OSPF Version 2", RFC 2328, April 1998.

[PACKETBB]
Clausen, T., Dearlove, C., Dean, J., and C. Adjih, "Generalized Mobile Ad Hoc Network (MANET) Packet/Message Format", RFC 5444, February 2009.

[RIP] Malkin, G., "RIP Version 2", RFC 2453, November 1998.

Appendix A. Cost and Metric Computation

The strategy for computing link costs and route metrics is a local matter; Babel itself only requires that it comply with the conditions given in Section 3.4.3 and Section 3.5.2. Different nodes may use different strategies in a single network and may use different strategies on different interface types. This section describes the strategies used by the sample implementation of Babel.

The sample implementation of Babel sends periodic Multicast Hellos, and never sends Unicast Hellos. It maintains statistics about the last 16 received Hello TLVs of each kind (Appendix A.1), computes costs by using the 2-out-of-3 strategy (Appendix A.2.1) on wired links, and ETX (Appendix A.2.2) on wireless links. It uses an additive algebra for metric computation (Appendix A.3.1).

A.1. Maintaining Hello History

For each neighbour, the sample implementation of Babel maintains two sets of Hello history, one for each kind of Hello, and an expected sequence number, one for Multicast and one for Unicast Hellos. Each Hello history is a vector of 16 bits, where a 1 value represents a received Hello, and a 0 value a missed Hello. For each kind of Hello, the expected sequence number, written ne, is the sequence number that is expected to be carried by the next received Hello from this neighbour.

Whenever it receives a Hello packet of a given kind from a neighbour, a node compares the received sequence number nr for that kind of Hello with its expected sequence number ne. Depending on the outcome of this comparison, one of the following actions is taken:

- if the two differ by more than 16 (modulo 2^16), then the sending node has probably rebooted and lost its sequence number; the whole associated neighbour table entry is flushed and a new one is created;

- otherwise, if the received nr is smaller (modulo 2^16) than the expected sequence number ne, then the sending node has increased its Hello interval without us noticing; the receiving node removes the last (ne - nr) entries from this neighbour's Hello history (we "undo history");

- otherwise, if nr is larger (modulo 2^16) than ne, then the sending node has decreased its Hello interval, and some Hellos were lost; the receiving node adds (nr - ne) 0 bits to the Hello history (we "fast-forward").

The receiving node then appends a 1 bit to the Hello history and sets ne to (nr + 1). If the Interval field of the received Hello is not zero, it resets the neighbour's hello timer to 1.5 times the advertised Interval (the extra margin allows for delay due to jitter).

Whenever either Hello timer associated to a neighbour expires, the local node adds a 0 bit to this neighbour's Hello history, and increments the expected Hello number. If both Hello histories are empty (they contain 0 bits only), the neighbour entry is flushed; otherwise, the relevant hello timer is reset to the value advertised in the last Hello of that kind received from this neighbour (no extra margin is necessary in this case, since jitter was already taken into account when computing the timeout that has just expired).

A.2. Cost Computation

This section discusses how to compute costs based on Hello history.

A.2.1. k-out-of-j

K-out-of-j link sensing is suitable for wired links that are either up, in which case they only occasionally drop a packet, or down, in which case they drop all packets.

The k-out-of-j strategy is parameterised by two small integers k and j, such that 0 < k <= j, and the nominal link cost, a constant K >= 1. A node keeps a history of the last j hellos; if k or more of those have been correctly received, the link is assumed to be up, and the rxcost is set to K; otherwise, the link is assumed to be down, and the rxcost is set to infinity.

Since Babel supports two kinds of Hellos, a Babel node performs k-out-of-j twice for each neighbour, once on the Unicast and once on the Multicast Hello history. If either of the instances of k-out-of-j indicates that the link is up, then the link is assumed to be up, and the rxcost is set to K; if both instances indicate that the link is down, then the link is assumed to be down, and the rxcost is set to infinity. In other words, the resulting rxcost is the minimum of the rxcosts yielded by the two instances of k-out-of-j link sensing.

The cost of a link performing k-out-of-j link sensing is defined as follows:

o cost = FFFF hexadecimal if rxcost = FFFF hexadecimal;

o cost = txcost otherwise.

A.2.2. ETX

Unlike wired links, which are bimodal (either up or down), wireless links exhibit continuous variation of the link quality. Naive application of hop-count routing in networks that use wireless links for transit tends to select long, lossy links in preference to shorter, lossless links, which can dramatically reduce throughput. For that reason, a routing protocol designed to support wireless links must perform some form of link-quality estimation.

ETX [ETX] is a simple link-quality estimation algorithm that is designed to work well with the IEEE 802.11 MAC. By default, the IEEE 802.11 MAC performs ARQ and rate adaptation on unicast frames, but not on multicast frames, which are sent at a fixed rate with no ARQ; therefore, measuring the loss rate of multicast frames yields a useful estimate of a link's quality.

A node performing ETX link quality estimation uses a neighbour's Multicast Hello history to compute an estimate, written beta, of the probability that a Hello TLV is successfully received. Beta can be computed as the fraction of 1 bits within a small number (say, 6) of the most recent entries in the Multicast Hello history, or it can be an exponential average, or some combination of both approaches.

Let alpha be MIN(1, 256/txcost), an estimate of the probability of successfully sending a Hello TLV. The cost is then computed by

```
cost = 256/(alpha * beta)
```

or, equivalently,

```
cost = (MAX(txcost, 256) * rxcost) / 256.
```

Since the IEEE 802.11 MAC performs ARQ on unicast frames, unicast frames do not provide a useful measure of link quality, and therefore ETX ignores the Unicast Hello history. Thus, a node performing ETX link-quality estimation will not route through neighbouring nodes unless they send periodic Multicast Hellos (possibly in addition to Unicast Hellos).

A.3. Metric Computation

As described in Section 3.5.2, the metric advertised by a neighbour is combined with the link cost to yield a metric.

A.3.1. Additive Metrics

The simplest approach for obtaining a monotonic, left-distributive metric is to define the metric of a route as the sum of the costs of the component links. More formally, if a neighbour advertises a route with metric m over a link with cost c, then the resulting route has metric M(c, m) = c + m.

A multiplicative metric can be converted into an additive one by taking the logarithm (in some suitable base) of the link costs.

A.3.2. External Sources of Willingness

A node may want to vary its willingness to forward packets by taking into account information that is external to the Babel protocol, such as the monetary cost of a link, the node's battery status, CPU load, etc. This can be done by adding to every route's metric a value k that depends on the external data. For example, if a battery-powered node receives an update with metric m over a link with cost c, it might compute a metric M(c, m) = k + c + m, where k depends on the battery status.

In order to preserve strict monotonicity (Section 3.5.2), the value k must be greater than -c.

Appendix B. Constants

The choice of time constants is a trade-off between fast detection of mobility events and protocol overhead. Two implementations of Babel with different time constants will interoperate, although the resulting convergence time will most likely be dictated by the slower of the two.

Experience with the sample implementation of Babel indicates that the Hello interval is the most important time constant: a mobility event is detected within 1.5 to 3 Hello intervals. Due to Babel's reliance on triggered updates and explicit requests, the Update interval only has an effect on the time it takes for accurate metrics to be propagated after variations in link costs too small to trigger an unscheduled update or in the presence of packet loss.

At the time of writing, the sample implementation of Babel uses the following default values:

Multicast Hello Interval: 4 seconds.

IHU Interval: the advertised IHU interval is always 3 times the Multicast Hello interval. IHUs are actually sent with each Hello on lossy links (as determined from the Hello history), but only with every third Multicast Hello on lossless links.

Unicast Hello Interval: the sample implementation never sends scheduled Unicast Hellos;

Update Interval: 4 times the Multicast Hello interval.

IHU Hold Time: 3.5 times the advertised IHU interval.

Route Expiry Time: 3.5 times the advertised update interval.

Source GC time: 3 minutes.

Request timeout: initially 2 seconds, doubled every time a request is resent, up to a maximum of three times.

The amount of jitter applied to a packet depends on whether it contains any urgent TLVs or not (Section 3.1). Urgent triggered updates and urgent requests are delayed by no more than 200ms; acknowledgments, by no more than the associated deadline; and other TLVs by no more than one-half the Multicast Hello interval.

Appendix C. Considerations for protocol extensions

Babel is an extensible protocol, and this document defines a number of mechanisms that can be used to extend the protocol in a backwards compatible manner:

- increasing the version number in the packet header;
- defining new TLVs;
- defining new sub-TLVs (with or without the mandatory bit set);
- defining new AEs;
- using the packet trailer.

This appendix is intended to guide designers of protocol extensions in chosing a particular encoding.

The version number in the Babel header should only be increased if the new version is not backwards compatible with the original protocol.

In many cases, an extension could be implemented either by defining a new TLV, or by adding a new sub-TLV to an existing TLV. For example, an extension whose purpose is to attach additional data to route updates can be implemented either by creating a new "enriched" Update TLV, by adding a non-mandatory sub-TLV to the Update TLV, or by adding a mandatory sub-TLV.

The various encodings are treated differently by implementations that do not understand the extension. In the case of a new TLV or of a sub-TLV with the mandatory bit set, the whole TLV is ignored by implementations that do not implement the extension, while in the case of a non-mandatory sub-TLV, the TLV is parsed and acted upon, and only the unknown sub-TLV is silently ignored. Therefore, a non-mandatory sub-TLV should be used by extensions that extend the Update in a compatible manner (the extension data may be silently ignored), while a mandatory sub-TLV or a new TLV must be used by extensions that make incompatible extensions to the meaning of the TLV (the whole TLV must be thrown away if the extension data is not understood).

Experience shows that the need for additional data tends to crop up in the most unexpected places. Hence, it is recommended that extensions that define new TLVs should make them self-terminating, and allow attaching sub-TLVs to them.

Adding a new AE is essentially equivalent to adding a new TLV: Update TLVs with an unknown AE are ignored, just like unknown TLVs. However, adding a new AE is more involved than adding a new TLV, since it creates a new set of compression state. Additionally, since the Next Hop TLV creates state specific to a given address family, as opposed to a given AE, a new AE for a previously defined address family must not be used in the Next Hop TLV if backwards compatibility is required. A similar issue arises with Update TLVs with unknown AEs establishing a new router-id (due to the Router-Id flag being set). Therefore, defining new AEs must be done with care if compatibility with unextended implementations is required.

The packet trailer is intended to carry cryptographic signatures that only cover the packet body; storing the cryptographic signatures in the packet trailer avoids clearing the signature before computing a hash of the packet body, and makes it possible to check a cryptographic signature before running the full, stateful TLV parser. Hence, only TLVs that don’t need to be protected by cryptographic security protocols should be allowed in the packet trailer. Any such TLVs should be easy to parse, and in particular should not require stateful parsing.

Appendix D. Stub Implementations

Babel is a fairly economic protocol. Updates take between 12 and 40 octets per destination, depending on the address family and how successful compression is; in a double-stack flat network, an average of less than 24 octets per update is typical. The route table occupies about 35 octets per IPv6 entry. To put these values into perspective, a single full-size Ethernet frame can carry some 65 route updates, and a megabyte of memory can contain a 20000-entry route table and the associated source table.

Babel is also a reasonably simple protocol. The sample implementation consists of less than 12 000 lines of C code, and it compiles to less than 120 kB of text on a 32-bit CISC architecture; about half of this figure is due to protocol extensions and user-interface code.

Nonetheless, in some very constrained environments, such as PDAs, microwave ovens, or abacuses, it may be desirable to have subset implementations of the protocol.

There are many different definitions of a stub router, but for the needs of this section a stub implementation of Babel is one that announces one or more directly attached prefixes into a Babel network but doesn’t reannounce any routes that it has learnt from its neighbours. It may either maintain a full routing table, or simply select a default gateway amongst any one of its neighbours that announces a default route. Since a stub implementation never forwards packets except from or to directly attached links, it cannot possibly participate in a routing loop, and hence it need not evaluate the feasibility condition or maintain a source table.

No matter how primitive, a stub implementation MUST parse sub-TLVs attached to any TLVs that it understands and check the mandatory bit. It MUST answer acknowledgment requests and MUST participate in the Hello/IHU protocol. It MUST also be able to reply to seqno requests for routes that it announces and SHOULD be able to reply to route requests.

Experience shows that an IPv6-only stub implementation of Babel can be written in less than 1000 lines of C code and compile to 13 kB of text on 32-bit CISC architecture.

Appendix E. Software Availability

The sample implementation of Babel is available from <https://www.irif.fr/˜jch/software/babel/>.

Appendix F. Changes from previous versions

F.1. Changes since RFC 6126

- Changed UDP port number to 6696.
- Consistently use router-id rather than id.
- Clarified that the source garbage collection timer is reset after sending an update even if the entry was not modified.
- In section "Seqno Requests", fixed an erroneous "route request".
- In the description of the Seqno Request TLV, added the description of the Router-Id field.
- Made router-ids all-0 and all-1 forbidden.

F.2. Changes since draft-ietf-babel-rfc6126bis-00

- Added security considerations.

F.3. Changes since draft-ietf-babel-rfc6126bis-01

- Integrated the format of sub-TLVs.
- Mentioned for each TLV whether it supports sub-TLVs.
- Added Appendix C.
- Added a mandatory bit in sub-TLVs.
- Changed compression state to be per-AF rather than per-AE.
- Added implementation hint for the routing table.
- Clarified how router-ids are computed when bit 0x40 is set in Updates.
- Relaxed the conditions for sending requests, and tightened the conditions for forwarding requests.
- Clarified that neighbours should be acquired at some point, but it doesn’t matter when.

F.4. Changes since draft-ietf-babel-rfc6126bis-02

- Added Unicast Hellos.
- Added unscheduled (interval-less) Hellos.
- Changed Appendix A to consider Unicast and unscheduled Hellos.
- Changed Appendix B to agree with the reference implementation.
- Added optional algorithm to avoid the hold time.
- Changed the table of pending seqno requests to be indexed by router-id in addition to prefixes.
- Relaxed the route acquisition algorithm.
- Replaced minimal implementations by stub implementations.
- Added acknowledgments section.

F.5. Changes since draft-ietf-babel-rfc6126bis-03

- Clarified that all the data structures are conceptual.
- Made sending and receiving Multicast Hellos a SHOULD, avoids expressing any opinion about Unicast Hellos.
- Removed opinion about Multicast vs. Unicast Hellos (Appendix A.4).
- Made hold-time into a SHOULD rather than MUST.
- Clarified that Seqno Requests are for a finite-metric Update.
- Clarified that sub-TLVs Pad1 and PadN are allowed within any TLV that allows sub-TLVs.
- Updated IANA Considerations.
- Updated Security Considerations.
- Renamed routing table back to route table.
- Made buffering outgoing updates a SHOULD.
- Weakened advice to use modified EUI-64 in router-ids.
- Added information about sending requests to Appendix B.

- A number of minor wording changes and clarifications.

F.6. Changes since draft-ietf-babel-rfc6126bis-03

Minor editorial changes.

F.7. Changes since draft-ietf-babel-rfc6126bis-04

- Renamed isotonicity to left-distributivity.
- Minor clarifications to unicast hellos.
- Updated requirements boilerplate to RFC 8174.
- Minor editorial changes.

F.8. Changes since draft-ietf-babel-rfc6126bis-05

- Added information about the packet trailer, now that it is used by draft-ietf-babel-hmac.

F.9. Changes since draft-ietf-babel-rfc6126bis-06

- Added references to security documents.

Authors' Addresses

Juliusz Chroboczek
IRIF, University of Paris-Diderot
Case 7014
75205 Paris Cedex 13
France

Email: jch@irif.fr

David Schinazi
Google LLC
1600 Amphitheatre Parkway
Mountain View, California 94043
USA

Email: dschinazi.ietf@gmail.com

```
Network Working Group                                         M. Boutier
Internet-Draft                                             J. Chroboczek
Updates: 6126bis (if approved)        IRIF, University of Paris-Diderot
Intended status: Standards Track                         August 21, 2017
Expires: February 22, 2018
```

Source-Specific Routing in Babel
draft-ietf-babel-source-specific-01

Abstract

Source-specific routing is an extension to traditional next-hop routing where packets are forwarded according to both their destination and their source address. This document describes the source-specific routing extension to the Standard Track's Babel routing protocol defined in [BABEL]. It is incompatible with the Experimental Track's Babel [RFC6126].

Source-specific routing is also known as Source Address Dependent Routing, SAD Routing, SADR, Destination/Source Routing or Source/ Destination Routing.

Status of This Memo

Copyright Notice

Table of Contents

1. TODOs

- Source Prefix sub-TLV type: TBD

- check references (Section) for BABEL in 6126bis

2. Introduction and background

The Babel routing protocol as defined is [BABEL] is a distance vector routing protocol for next-hop routing. In next-hop routing, each node maintains a forwarding table which maps prefixes to next-hops. The forwarding decision is a per-packet operation which depends on the destination address of the packets and on the entries of the forwarding table. When a packet is about to be routed, its destination address is compared to the prefixes of the routing table: the entry with the most specific prefix containing the destination address of the packet is choosen, and the packet is forwarded to the associated next-hop. Next-hop routing is a simple, well understood paradigm that works satisfactorily in a large number of cases.

Source-specific routing is a modest extension of next-hop routing where the forwarding decision additionnaly depends on the source address of the packets. The forwarding tables are extended to map pairs of prefixes (destination, source) to a next-hop. When multiple entries are candidate to route a packet, the one with the most specific destination prefix is choosen, and in case of equality the one with the most specific source. In source-specific routing, two packets with the same destination but different sources may be forwarded among different paths.

The main application of source-specific routing is, at the time of this writing, multihoming with Provider Agregatable (PA) addresses. In such configuration, each Internet Service Provider (ISP) provides to the network a PA prefix and a default route for this prefix while performing ingress filtering ([BCP84]). Each host has one address per ISP, and sends packets with one of these addresses as source address. Source-specific routing ensures that packets are routed towards the provider of their source address, such that they are not filtered out. More details and more use cases can be found in [SS-ROUTING],[IETF-SSR].

This document describes the source-specific routing extension for the Babel routing protocol [BABEL]. This involves changes to data structures and protocol messages. The data structures receive an additionnal source prefix which is part of the index, similarly to (and with) the destination prefix. The Update, Route Request and Seqno Request are the three messages which carry a (destination) prefix: they are extended with a source prefix.

3. Data Structures

Some of the data structures of a Babel node contains a destination prefix or are partly indexed by a destination prefix. This extension adds a source prefix to these structures and indexes.

3.1. The Source Table

Every Babel node maintains a source table, as described in [BABEL], Section 3.2.5. A source-specific Babel node extends this table with the following field. With this extension, the source table is indexed by triples of the form (prefix, source prefix, router-id).

- the source prefix specifying the source address of packets to which this entry applies.

If a source table entry has a zero length source prefix, then the entry is a non-source-specific entry, and is treated just like a source table entry defined by the original Babel protocol.

With this extension, the route entry contains a source which itself contains a source prefix. These are two very different concepts, and should not be confused.

3.2. The Route Table

Every Babel node maintains a route table, as described in [BABEL], Section 3.2.6. With this extension, the route table is indexed by triples of the form (prefix, source prefix, neighbour) obtained from the associated source table entry.

If a route table entry has a zero length source prefix, then the entry is a non-source-specific entry, and is treated just like a route table entry defined by the original Babel protocol.

3.3. The Table of Pending Seqno Requests

Every Babel node maintains a table of pending seqno requests, as described in [BABEL], Section 3.2.7. A source-specific Babel node extends this table with the following entry. With this extension, the table of pending seqno requests is indexed by triples of the form (prefix, source prefix, router-id).

- the source prefix being requested.

4. Data Forwarding

In next-hop routing, if two routing table entries overlap, then one is necessarily more specific than the other; the "longest prefix rule" specifies that the most specific applicable routing table entry is chosen.

With source-specific routing, there might no longer be a most specific applicable entry: two routing table entries might match a given packet without one necessarily being more specific than the other. Consider for example the following routing table:

destination	source	next-hop
2001:DB8:0:1::/64	::/0	A
::/0	2001:DB8:0:2::/64	B

This specifies that all packets with destination in 2001:DB8:0:1::/64 are to be routed through A, while all packets with source in 2001:DB8:0:2::/64 are to be routed through B. A packet with source 2001:DB8:0:2::42 and destination 2001:DB8:0:1::57 matches both rules, although neither is more specific than the other. A choice is necessary, and unless the choice being made is the same on all routers in a routing domain, persistent routing loops may occur. More informations are available in [SS-ROUTING] Section IV.C.

A Babel implementation MUST choose routing table entries by using the so-called destination-first ordering, where a routing table entry R1 is preferred to a routing table entry R2 when either R1's destination prefix is more specific than R2's, or the destination prefixes are equal and R1's source prefix is more specific than R2's. (In more formal terms, routing table entries are compared using the lexicographic product of the destination prefix ordering by the source prefix ordering.)

In practice, this means that a source-specific Babel implementation must take care that any lower layer that performs packet forwarding obey this semantics. In particular:

- If the lower layers implement the destination-first ordering, then the Babel implementation MAY use them directly;

- If the lower layers can hold source-specific routes, but not with the right semantics, then the Babel implementation MUST disambiguate the routing table by using a suitable disambiguation algorithm (see [SS-ROUTING] Section V.B for such an algorithm);

- If the lower layers cannot hold source-specific routes, then a Babel implementation MUST silently ignore (drop) any source-specific routes.

5. Protocol Operation

This extension does not fundamentally change the operation of the Babel protocol. We only describe the fundamental differences between the original protocol and this extension in this section. The other mechanisms described in [BABEL] (Section 3) are extended to pairs of (destination, source) prefixes instead of just (destination) prefixes.

5.1. Source-specific messages

Three messages carry a destination prefix: Updates, Route Requests and Seqno Requests. These messages are extended to carry, in addition, a source prefix if (and only if) the corresponding route is source-specific. More formally, an Update, a Route Request and a Seqno Request MUST carry a source prefix if they concern a source-specific route (non-zero length source prefix) and MUST NOT carry a source prefix otherwise (zero length source prefix). A message which carries a source prefix is said to be source-specific.

5.2. Route Acquisition

When a non-source-specific Babel node receives a source-specific update, it silently ignores it. When a source-specific Babel node receives a non-source-specific update, it MUST treat this update as a zero length source-specific update.

When a source-specific Babel node receives a source-specific update (prefix, source prefix, router-id, seqno, metric) from a neighbour neigh, it behaves as described in [BABEL] (Section 3.5.4) though indexing entries by (prefix, source prefix, neigh).

5.3. Wildcard retractions (update)

The original protocol defines a wildcard update with AE equals to 0 as being a wildcard retraction. A node receiving a wildcard retraction on an interface must consider that the sending node retracts all the routes it advertised on this interface.

Wildcard retractions are used when a node is about to leave the network. Thus, this extension does not define source-specific wildcard retraction, but extends wildcard retraction to apply also to source-specific routes. More formally, a wildcard update MUST NOT carry a source prefix, and a source-specific Babel node receiving a (legacy) wildcard update MUST retracts all routes it learns from this node (including source-specific ones).

5.4. Wildcard requests

The original Babel protocol states that when a node receives a wildcard route request, it SHOULD send a full routing table dump. This extension does not change this statement: a source-specific node SHOULD send a full routing table dump when receiving a wildcard request.

Source-specific wildcard requests does not exist: a wildcard request MUST NOT carry a source prefix, and a source prefix associated with a wildcard update SHOULD be ignored.

One of the motivation behalf this design choice is that wildcard requests are defined with AE equals to 0. They naturally apply to AE 1, AE 2 and AE 3 defined in [BABEL], but also to any other AE which may be defined in the future. New AEs, new TLVs or new sub-TLVs are extension mechanisms. Thus, the semantics of a wildcard request is clearly to also asks for routes coming from extensions.

6. Compatibility with the base protocol

The protocol extension defined in this document is, to a great extent, interoperable with the base protocol defined in [BABEL] (and all its known extensions). More precisely, if non-source-specific routers and source-specific routers are mixed in a single routing domain, Babel's loop-avoidance properties are preserved, and, in particular, no persistent routing loops will occur.

However, this extension is not compatible with the Experimental Track's Babel Routing Protocol [RFC6126]. It requires the mandatory sub-TLV introduced in [BABEL]. Consequently, this extension MUST NOT be used with routers implementing RFC 6126, otherwise persistent routing loops may occur.

6.1. Loop-avoidance

The extension defined in this protocol uses a new Mandatory sub-TLV to carry the source prefix information. As discussed in Section 4.4 of [BABEL], this encoding ensures that non-source-specific routers will silently ignore the whole TLV, which is necessary to avoid persistent routing loops in hybrid networks.

Consider two nodes A and B, with A source-specific announcing a route to (D, S). Suppose that B (non source-specific) merely ignores the source prefix information when it receives the update rather than ignoring the whole TLV, and reannounces the route as D. This reannouncement reaches 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, causing a persistent routing loop:

```
       (D,S)                 (D)
        <--                 <--
     ------ A ----------------- B
              -->
             (D,::/0)
```

6.2. Starvation and Blackholes

In general, discarding source-specific routes by non-source-specific routers will cause route starvation. Intuitively, unless there are enough non-source-specific routes in the network, non-source-specific routers will suffer starvation, and discard packets for destinations that are only announced by source-specific routers.

A simple yet sufficient condition for avoiding starvation is to build a connected source-specific backbone that includes all of the edge routers, and announce a (non-source-specific) default route towards the backbone.

7. Protocol Encoding

This extension defines a new sub-TLV used to carry a source prefix by the three following existing messages: Update, Route Request and Seqno Request.

7.1. Source Prefix sub-TLV

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Type = TBD[128]|    Length     |  Source Plen  | Source Prefix...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
```

Fields:

Type   Set to TBD[128] to indicate a Source Prefix sub-TLV.

Length   The length of the body, exclusive of the Type and Length fields.

Source Plen The length of the advertised source prefix. This MUST NOT be 0.

Source Prefix The source prefix being advertised. This field’s size is (Source Plen)/8 rounded upwards.

The source prefix encoding (AE) is the same as the Prefix’s. It is defined by the AE field of the corresponding TLV.

Note that this sub-TLV is a Mandatory sub-TLV. The whole TLV MUST be ignored if that sub-TLV is not recognized. Otherwise, routing loops may occur (see Section 6.1).

7.2. Source-specific Update

The source-specific Update is an Update TLV with a Source Prefix sub-TLV. It advertises or retracts source-specific routes in the same manner than routes with non-source-specific Updates (see [BABEL]). A wildcard retraction (Update with AE equals to 0) MUST NOT carry a Source Prefix sub-TLV.

Contrary to the destination prefix, this extension does not compress the source prefix attached to Updates. However, as defined in [BABEL] (Section 4.5), the compression is allowed for the destination prefix of source-specific routes. Legacy implementation will correctly update their parser state while ignoring the whole TLV afterwards.

7.3. Source-specific (Route) Request

A source-specific Route Request is a Route Request TLV with a Source Prefix sub-TLV. It prompts the receiver to send an update for a given pair of destination and source prefixes. A wildcard request (Route Request with AE equals to 0) MUST NOT carry a Source Prefix sub-TLV.

7.4. Source-Specific Seqno Request

A source-specific Seqno Request is a Seqno Request TLV with a Source Prefix sub-TLV. It is just like a Seqno Request for a source-specific route. It uses the same mechanisms described in [BABEL].

8. IANA Considerations

IANA is requested to allocate TBD, a Babel sub-TLV type from the range reserved for mandatory sub-TLVs [value 128 suggested], and to add the following entry to the "Babel mandatory sub-TLV Types" registry:

Type	Name	Reference
TBD[128]	Source Prefix	(this document)

9. Security considerations

The extension defined in this document adds a new sub-TLV to three TLVs already present in the original Babel protocol. It does not by itself change the security properties of the protocol.

10. References

10.1. Normative References

[BABEL] Chroboczek, J., "The Babel Routing Protocol", Internet Draft draft-ietf-babel-rfc6126bis-02, May 2017.

[BCP84] Baker, F. and P. Savola, "Ingress Filtering for Multihomed Networks", BCP 84, RFC 3704, March 2004.

[IETF-SSR] Lamparter, D. and A. Smirnov, "Destination/Source Routing", Internet Draft draft-ietf-rtgwg-dst-src-routing, May 2017.

[RFC6126] Chroboczek, J., "The Babel Routing Protocol (Experimental)", RFC 6126, February 2011.

10.2. Informative References

[SS-ROUTING] Boutier, M. and J. Chroboczek, "Source-Specific Routing", August 2014.

In Proc. IFIP Networking 2015. A slightly earlier version is available online from http://arxiv.org/pdf/1403.0445.

Authors' Addresses

Matthieu Boutier
IRIF, University of Paris-Diderot
Case 7014
75205 Paris Cedex 13
France

Email: boutier@irif.fr

Juliusz Chroboczek
IRIF, University of Paris-Diderot
Case 7014
75205 Paris Cedex 13
France

Email: jch@irif.fr

Network Working Group M. Boutier
Internet-Draft J. Chroboczek
Intended status: Standards Track IRIF, University of Paris-Diderot
Expires: April 26, 2019 October 23, 2018

Source-Specific Routing in Babel
draft-ietf-babel-source-specific-04

Abstract

Source-specific routing (also known as Source-Address Dependent Routing, SADR) is an extension to traditional next-hop routing where packets are forwarded according to both their destination and their source address. This document describes an extension for source-specific routing to the Babel routing protocol.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction and background

The Babel routing protocol [BABEL] is a distance vector routing protocol for next-hop routing. In next-hop routing, each node maintains a forwarding table which maps destination prefixes to next hops. The forwarding decision is a per-packet operation which depends on the destination address of the packets and on the entries of the forwarding table. When a packet is about to be routed, its destination address is compared to the prefixes of the routing table: the entry with the most specific prefix containing the destination address of the packet is chosen, and the packet is forwarded to the associated next-hop. Next-hop routing is a simple, well understood paradigm that works satisfactorily in a large number of cases.

Source-specific routing [SS-ROUTING], or Source Address Dependent Routing (SADR) [DSR], is a modest extension to next-hop routing where the forwarding decision depends not only on the destination address but also on the source address of the packet being routed, which makes it possible for two packets with the same destination but different source addresses to be routed following different paths.

The forwarding tables are extended to map pairs of prefixes (destination, source) to next hops. When multiple entries match a given packet, the one with the most specific destination prefix is chosen, and, in case of equality, the one with the most specific source prefix.

The main application of source-specific routing is a form of multihoming known as multihoming with multiple addresses. When using this technique in a network connected to multiple providers, every host is assigned multiple addresses, one per provider. When a host sources a packet, it picks one of its addresses as the source address, and source-specific routing is used to route the packet to an edge router that is connected to the corresponding provider, which is compatible with [BCP84]. Unlike classical multihoming, this technique is applicable to small networks, as it does not require the use of provider-independent addresses, or cause excessive growth of the global routing table. More details are given in [SS-ROUTING] and [DSR].

This document describes a source-specific routing extension for the Babel routing protocol [BABEL]. This involves minor changes to the data structures, which must include a source prefix in addition to the destination prefix already present, and some changes to the Update, Route Request and Seqno Request TLVs, which are extended with a source prefix. The source prefix is encoded using a mandatory sub-TLV ([BABEL] Section 4.4).

2. Specification of Requirements

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Data Structures

A number of the conceptual data structures described in Section 3.2 of [BABEL] contain a destination prefix. This specification extends these data structures with a source prefix. Data from the original protocol, which do not specify a source prefix, are stored with a zero length source prefix, which matches exactly the same set of packets as the original, non-source-specific data.

3.1. The Source Table

Every Babel node maintains a source table, as described in [BABEL] Section 3.2.5. A source-specific Babel node extends this table with the following field:

- The source prefix specifying the source address of packets to which this entry applies.

The source table is now indexed by triples of the form (prefix, source prefix, router-id).

Note that the route entry contains a source (see sections 2 and 3.2.5 of [BABEL]) which itself contains a source prefix. These are two very different concepts that should not be confused.

3.2. The Route Table

Every Babel node maintains a route table, as described in [BABEL] Section 3.2.6. Each route table entry contains, among other data, a source, which this specification extends with a source prefix as described above. The route table is now indexed by triples of the form (prefix, source prefix, neighbour), where the prefix and source prefix are obtained from the source.

3.3. The Table of Pending Seqno Requests

Every Babel node maintains a table of pending seqno requests, as described in [BABEL], Section 3.2.7. A source-specific Babel node extends this table with the following entry:

- The source prefix being requested.

The table of pending seqno requests is now indexed by triples of the form (prefix, source prefix, router-id).

4. Data Forwarding

In next-hop routing, if two routing table entries overlap, then one is necessarily more specific than the other; the "longest prefix rule" specifies that the most specific applicable routing table entry is chosen.

With source-specific routing, there might no longer be a most specific applicable entry: two routing table entries might match a given packet without one necessarily being more specific than the other. Consider for example the following routing table:

destination	source	next-hop
2001:DB8:0:1::/64	::/0	A
::/0	2001:DB8:0:2::/64	B

This specifies that all packets with destination in 2001:DB8:0:1::/64 are to be routed through A, while all packets with source in 2001:DB8:0:2::/64 are to be routed through B. A packet with source 2001:DB8:0:2::42 and destination 2001:DB8:0:1::57 matches both rules, although neither is more specific than the other. A choice is necessary, and unless the choice being made is the same on all routers in a routing domain, persistent routing loops may occur. More details are given in Section IV.C of [SS-ROUTING].

A Babel implementation MUST choose routing table entries by using the so-called destination-first ordering, where a routing table entry R1 is preferred to a routing table entry R2 when either R1’s destination prefix is more specific than R2’s, or the destination prefixes are equal and R1’s source prefix is more specific than R2’s. (In more formal terms, routing table entries are compared using the lexicographic product of the destination prefix ordering by the source prefix ordering.) This is consistent with the behaviour described in Section 3.3 of [DSR].

In practice, this means that a source-specific Babel implementation must take care that any lower layer that performs packet forwarding obey this semantics. More precisely:

- If the lower layers implement the destination-first ordering, then the Babel implementation MAY use them directly;

- If the lower layers can hold source-specific routes, but not with the right semantics, then the Babel implementation MUST disambiguate the routing table by using a suitable disambiguation algorithm (see Section V.B of [SS-ROUTING] for such an algorithm);

- If the lower layers cannot hold source-specific routes, then a Babel implementation MUST silently ignore (drop) any source-specific routes.

5. Protocol Operation

This extension does not fundamentally change the operation of the Babel protocol, and we therefore only describe differences between the original protocol and the extended protocol.

In the original protocol, three TLVs carry a destination prefix: Updates, Route Requests and Seqno Requests. This specification extends these messages to optionally carry a source prefix sub-TLV, as described in Section 7 below. The sub-TLV is marked as mandatory, so that an unextended implementation will silently ignore the whole enclosing TLV. A node obeying this specification MUST NOT send a TLV with a zero-length source prefix: instead, it sends a TLV with no source prefix sub-TLV. Conversely, an extended implementation MUST interpret an unextended TLV as carrying a source prefix of zero length. Taken together, these properties ensure interoperability between the original and extended protocols (see Section 6 below).

5.1. Protocol Messages

This extension allows three TLVs of the original Babel protocol to carry a source prefix: Update TLVs, Route Request TLVs and Seqno Request TLVs.

In order to advertise a route with a non-zero length source prefix, a node sends a source-specific Update, i.e., an Update with a source prefix sub-TLV. When a node receives a source-specific Update (prefix, source prefix, router-id, seqno, metric) from a neighbour neigh, it behaves as described in [BABEL] Section 3.5.4, except that the entry under consideration is indexed by (prefix, source prefix, neigh) rather than just (prefix, neigh).

Similarly, when a node needs to send a Request of either kind that applies to a route with a non-zero length source prefix, it sends a source-specific Request, i.e., a Request with a source prefix sub-TLV. When a node receives a source-specific Request, it behaves as described in Section 3.8 of [BABEL], except that the request applies to the Route Table entry carrying the source prefix indicated by the sub-TLV.

5.2. Wildcard Messages

In the original protocol, the Address Encoding value 0 is used for wildcard messages: messages that apply to all routes, of any address family and with any destination prefix. Wildcard messages are allowed in two places in the protocol: wildcard retractions are used to retract all of the routes previously advertised by a node on a given interface, and wildcard Route Requests are used to request a full dump of the Route Table from a given node. Wildcard messages are intended to apply to all routes, including routes decorated with additional data and AE values to be defined by future extensions, and hence this specification extends wildcard operations to apply to all routes, whatever the value of the source prefix.

More precisely, a node receiving an Update with the AE field set to 0 and the Metric field set to infinity (a wildcard retraction) MUST apply the route acquisition procedure described in Section 3.5.4 of

[BABEL] to all of the routes that is has learned from the sending node, whatever the value of the source prefix. A node MUST NOT send a wildcard retraction with an attached source prefix, and a node that receives a wildcard retraction with a source prefix MUST ignore it.

Similarly, a node that receives a route request with the AE field set to 0 (a wildcard route request) SHOULD send a full routing table dump, including routes with a non-zero length source prefix. A node MUST NOT send a wildcard request that carries a source prefix, and a node receiving a wildcard request with a source prefix MUST ignore it.

6. Compatibility with the base protocol

The protocol extension defined in this document is, to a great extent, interoperable with the base protocol defined in [BABEL] (and all of its extensions). More precisely, if non-source-specific routers and source-specific routers are mixed in a single routing domain, Babel's loop-avoidance properties are preserved, and, in particular, no persistent routing loops will occur.

However, this extension is encoded using mandatory sub-TLVs, introduced in [BABEL], and therefore is not compatible with the older version of the Babel Routing Protocol [RFC6126] which does not support such sub-TLVs. Consequently, this extension MUST NOT be used with routers implementing RFC 6126, otherwise persistent routing loops may occur.

6.1. Loop-avoidance

The extension defined in this protocol uses a new Mandatory sub-TLV to carry the source prefix information. As discussed in Section 4.4 of [BABEL], this encoding ensures that non-source-specific routers will silently ignore the whole TLV, which is necessary to avoid persistent routing loops in hybrid networks.

Consider two nodes A and B, with A source-specific announcing a route to (D, S). Suppose that B (non-source-specific) merely ignores the source prefix information when it receives the update rather than ignoring the whole TLV, and re-announces the route as D. This re-announcement reaches 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, causing a persistent routing loop:

```
       (D,S)                 (D)
        <--               <--
     ------ A ----------------- B
               -->
             (D,::/0)
```

6.2. Starvation and Blackholes

In general, discarding source-specific routes by non-source-specific routers will cause route starvation. Intuitively, unless there are enough non-source-specific routes in the network, non-source-specific routers will suffer starvation, and discard packets for destinations that are only announced by source-specific routers.

A simple yet sufficient condition for avoiding starvation is to build a connected source-specific backbone that includes all of the edge routers, and announce a (non-source-specific) default route towards the backbone.

7. Protocol Encoding

This extension defines a new sub-TLV used to carry a source prefix: the Source Prefix sub-TLV. It can be used within an Update, a Route Request or a Seqno Request TLV to match a source-specific entry of the Route Table, in conjunction with the destination prefix natively carried by these TLVs.

Since a source-specific routing entry is characterized by a single destination prefix and a single source prefix, a source-specific message contains exactly one Source Prefix sub-TLV. A node MUST NOT send more that one Source Prefix sub-TLV in a TLV, and a node receiving more than one Source Prefix sub-TLV in a single TLV SHOULD ignore this TLV. It MAY ignore the whole packet.

7.1. Source Prefix sub-TLV

```
    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |   Type = 128  |    Length     |  Source Plen  | Source Prefix...
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
```

Fields:

Type   Set to 128 to indicate a Source Prefix sub-TLV.

Length   The length of the body, exclusive of the Type and Length fields.

Source Plen The length of the advertised source prefix. This MUST NOT be 0.

Source Prefix The source prefix being advertised. This field's size is (Source Plen)/8 rounded upwards.

The contents of the source prefix sub-TLV are interpreted according to the AE of the enclosing TLV.

Note that this sub-TLV is a mandatory sub-TLV. Therefore, as described in Section 4.4 of [BABEL], the whole TLV MUST be ignored if that sub-TLV is not understood (or malformed). Otherwise, routing loops may occur (see Section 6.1).

7.2. Source-specific Update

The source-specific Update is an Update TLV with a Source Prefix sub-TLV. It advertises or retracts source-specific routes in the same manner than routes with non-source-specific Updates (see [BABEL]). A wildcard retraction (Update with AE equal to 0) MUST NOT carry a Source Prefix sub-TLV.

Babel uses a stateful compression scheme to reduce the size taken by destination prefixes in update TLVs (see Section 4.5 of [BABEL]). The source prefix defined by this extension is not compressed. On the other hand, compression is allowed for the destination prefixes carried by source-specific updates. As described in Section 4.5 of [BABEL], unextended implementations will correctly update their parser state while otherwise ignoring the whole TLV.

7.3. Source-specific (Route) Request

A source-specific Route Request is a Route Request TLV with a Source Prefix sub-TLV. It prompts the receiver to send an update for a given pair of destination and source prefixes, as described in Section 3.8.1.1 of [BABEL]. A wildcard request (Route Request with AE equals to 0) MUST NOT carry a Source Prefix sub-TLV.

7.4. Source-Specific Seqno Request

A source-specific Seqno Request is a Seqno Request TLV with a Source Prefix sub-TLV. It requests the receiving node to perform the procedure described in Section 3.8.1.2 of [BABEL], but applied to a pair of a destination and source prefix.

8. IANA Considerations

IANA has allocated sub-TLV number 128 for the Source Prefix sub-TLV in the Babel sub-TLV types registry.

9. Security considerations

The extension defined in this document adds a new sub-TLV to three TLVs already present in the original Babel protocol, and does not in itself change the security properties of the protocol. However, source-specific routing gives more control over routing to the sending hosts, which might have security implications (see Section 8 of [DSR]).

10. Acknowledgments

The authors are grateful to Joel Halpern for his help with this document.

11. References

11.1. Normative References

[BABEL] Chroboczek, J., "The Babel Routing Protocol", Internet Draft draft-ietf-babel-rfc6126bis-06, October 2018.

[BCP84] Baker, F. and P. Savola, "Ingress Filtering for Multihomed Networks", BCP 84, RFC 3704, March 2004.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017.

11.2. Informative References

[DSR] Lamparter, D. and A. Smirnov, "Destination/Source Routing", Internet Draft draft-ietf-rtgwg-dst-src-routing-06, May 2018.

[RFC6126] Chroboczek, J., "The Babel Routing Protocol", RFC 6126, DOI 10.17487/RFC6126, April 2011, <https://www.rfc-editor.org/info/rfc6126>.

[SS-ROUTING]
Boutier, M. and J. Chroboczek, "Source-Specific Routing", August 2014.

In Proc. IFIP Networking 2015. A slightly earlier version is available online from http://arxiv.org/pdf/1403.0445.

Authors' Addresses

Matthieu Boutier
IRIF, University of Paris-Diderot
Case 7014
75205 Paris Cedex 13
France

Email: boutier@irif.fr

Juliusz Chroboczek
IRIF, University of Paris-Diderot
Case 7014
75205 Paris Cedex 13
France

Email: jch@irif.fr

BIER Z. Zhang
Internet-Draft ZTE Corporation
Intended status: Standards Track A. Przygienda
Expires: April 29, 2018 Juniper Networks, Inc.
October 26, 2017

BIER in IPv6
draft-zhang-bier-bierin6-00

Abstract

BIER is a new architecture for the forwarding of multicast data packets. This document defines native IPv6 encapsulation for BIER hop-by-hop forwarding or BIERin6 for short.

Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC2119.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

BIER [I-D.ietf-bier-architecture] is a new architecture for the forwarding of multicast data packets. It provides optimal forwarding through a "multicast domain' and it does not necessarily precondition construction of a multicast distribution tree, nor does it require intermediate nodes to maintain any per-flow state.

[I-D.ietf-bier-mpls-encapsulation] defines the BIER encapsulation format in MPLS and non-MPLS environment that rely on MPLS labels or special ethernet type support. In pure IPv6 environments a BIER packet could be forwarded by means of simple IPv6 hop-by-hop processing only, without any new hardware support. Ultimate hardware support is obviously possible but the encapsulation is especially interesting for environments like [RFC7368] where high throughput multicast forwarding performance is not decisive and could be initially done in slow-path or end host (assuming last-hop being IPv6 encapsulated).

This document defines native BIER IPv6 [RFC2460] encapsulation we call BIERv6. This encapsulation is aligned with the format defined in [I-D.ietf-bier-mpls-encapsulation] for a non-mpls version.

This document uses terminology defined in [I-D.ietf-bier-architecture].

2. IPv6 Header

The BIER packet itself is the payload of an IPv6 frame. The destination address field in IPv6 packet MAY be the neighbor's link-local or one of the loopback interface addresses of that neighbor. The destination address SHOULD be the BFR-prefix advertised by IGP/BGP extensions for BIER. TTL value of 1 MUST be used on the IPv6 packet.

The source address field in IPv6 packet MAY be the loopback interface address of the sending BFIR. The address SHOULD be the BFR-prefix advertised in IGP/BGP extension.

A new next-protocol type in IPv6 Next header field of TBD indicates the following BIER packet.

The Flow-ID in the IPv6 packet SHOULD be copied from the entropy field in the BIER encapsulation.

3. BIER Header

S bit in BIER header has no significance in this environment. It should be set to 1 upon transmission, but it MUST be ignored upon reception.

TC bits in BIER header have no significance in this environment since the IPv6 packet TC takes precedence on processing. It should be set to zero upon transmission, but it MUST be ignored upon reception.

The BIFT-id is used to indicate the combination of <SD, SI, BSL> normally; it should be set to the value advertised by the next-hop BFR through e.g. IGP [I-D.ietf-bier-ospf-bier-extensions], [I-D.zhang-bier-babel-extensions] or BGP [I-D.ietf-bier-idr-extensions] extension for BIER.

The remaining fields defined in BIER header MUST assume the same values and be afforded same treatement as specified in [I-D.ietf-bier-mpls-encapsulation].

4. IANA Considerations

IANA is requested to set up a new type of "Next header" registry value for BIERv6 in the "Assigned Internet Protocol Numbers" registry.

5. Security Considerations

General IPv6 and BIER security considerations apply.

6. References

6.1. Normative References

[I-D.ietf-bier-architecture]
Wijnands, I., Rosen, E., Dolganow, A., Przygienda, T., and S. Aldrin, "Multicast using Bit Index Explicit Replication", draft-ietf-bier-architecture-08 (work in progress), September 2017.

[I-D.ietf-bier-idr-extensions]
Xu, X., Chen, M., Patel, K., Wijnands, I., and T. Przygienda, "BGP Extensions for BIER", draft-ietf-bier-idr-extensions-03 (work in progress), August 2017.

[I-D.ietf-bier-mpls-encapsulation]
Wijnands, I., Rosen, E., Dolganow, A., Tantsura, J., Aldrin, S., and I. Meilik, "Encapsulation for Bit Index Explicit Replication in MPLS and non-MPLS Networks", draft-ietf-bier-mpls-encapsulation-10 (work in progress), October 2017.

[I-D.ietf-bier-ospf-bier-extensions]
Psenak, P., Kumar, N., Wijnands, I., Dolganow, A., Przygienda, T., Zhang, Z., and S. Aldrin, "OSPF Extensions for BIER", draft-ietf-bier-ospf-bier-extensions-09 (work in progress), October 2017.

[I-D.zhang-bier-babel-extensions]
Zhang, Z. and T. Przygienda, "BIER in BABEL", draft-zhang-bier-babel-extensions-01 (work in progress), June 2017.

[RFC2460] Deering, S. and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", RFC 2460, DOI 10.17487/RFC2460, December 1998, <https://www.rfc-editor.org/info/rfc2460>.

6.2. Informative References

[RFC7368] Chown, T., Ed., Arkko, J., Brandt, A., Troan, O., and J. Weil, "IPv6 Home Networking Architecture Principles", RFC 7368, DOI 10.17487/RFC7368, October 2014, <https://www.rfc-editor.org/info/rfc7368>.

Authors' Addresses

Zheng(Sandy) Zhang
ZTE Corporation

EMail: zzhang_ietf@hotmail.com

Tony Przygienda
Juniper Networks, Inc.

EMail: prz@juniper.net